

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE (DD-MM-YYYY)

22/03/2007

2. REPORT TYPE

Final Report

3. DATES COVERED (From - To)

1/1/2004 - 12/31/2006

4. TITLE AND SUBTITLE

Large Eddy Simulation Using a Transport Equation

for the Subgrid-Scale Stress Tensor

5a. CONTRACT NUMBER

5b. GRANT NUMBER

FA9550-04-1-0023

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Blair Perot

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Massachusetts

219 Elab

Amherst MA 01003

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFOSR, John Schmisser/NA

875 North Randolph Street

Suite 326

Arlington VA 22203-1768

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Public

Approved for Public Release - Dist A

AFRL-SR-AR-TR-07-0204

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The objective of this research was to demonstrate that classical transport equation (RANS) models can be applied at any mesh resolution. In particular, we show that transport equation models like the classical k/ϵ model also make excellent subgrid scale models for Large Eddy Simulation (LES). However, this research is not concerned with the development of a particular RANS/LES model but a general approach to turbulence modeling for any mesh resolution. To confirm the generality of the approach, a Reynolds stress transport (RST) equation model is also shown to work well as an automatically adaptive LES subgrid scale model. Unlike other hybrid modeling approaches that can address a range of mesh scales, the demonstrated approach is self-adaptive. It will always calculate using first principals as much of the turbulence as the mesh allows, and will model the rest. The character of the model is self-adjusting and is not a function of some external input such as the geometry. The approach is easy to implement using existing CFD codes.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:

a. REPORT

b. ABSTRACT

c. THIS PAGE

17. LIMITATION OF ABSTRACT

18. NUMBER OF PAGES

43

19a. NAME OF RESPONSIBLE PERSON

Blair Perot

19b. TELEPHONE NUMBER (include area code)

413-545-3925

Large Eddy Simulation Using a Transport Equation for the Subgrid-Scale Stress Tensor

FA9550-04-1-0023

Blair Perot

Department of Mechanical Engineering
University of Massachusetts, Amherst

Abstract

The objective of this research was to demonstrate that classical transport equation (RANS) models can be applied at any mesh resolution. In particular, we show that transport equation models like the classical k/ϵ model also make excellent subgrid scale models for Large Eddy Simulation (LES). However, this research is not concerned with the development of a particular RANS/LES model but a general approach to turbulence modeling for any mesh resolution. To confirm the generality of the approach, a Reynolds stress transport (RST) equation model is also shown to work well as an automatically adaptive LES subgrid scale model.

Unlike other hybrid modeling approaches that can address a range of mesh scales, the demonstrated approach is self-adaptive. It will always calculate using first principals as much of the turbulence as the mesh allows, and will model the rest. Unlike other hybrid models, this approach is not a blending of two models, nor does it require additional user specified constants. The character of the model is self-adjusting and is not a function of some external input such as the geometry. The approach is easy to implement using existing CFD codes.

Turbulence modeling is *the* bottleneck in current Computational Fluid Dynamics (CFD) predictions of engineering flows. The proposed modeling approach is fundamentally different from prior LES models and current hybrid models in that it achieves a completely natural evolution from RANS to LES to DNS, using largely existing modeling technology.

1. Introduction

Turbulence models are frequently classified by the ratio of how much turbulent energy is represented by the model compared to how much turbulent energy is computed via first principals. RANS (Reynolds averaged Navier-Stokes) models represent the most turbulent energy in the model. LES (large eddy simulation) computes considerably more of the turbulent energy via first principals and DNS (direct numerical simulation) computes all the turbulent energy correctly and models none. If a more detailed terminology is desired, in between RANS and LES lies URANS (unsteady RANS) and VLES (very large eddy simulation). The range of these turbulence models is shown in Figure 1 in relation to a 3D turbulent energy spectrum. Each model, tries to represent the energy in the spectrum to the right of the model's name.

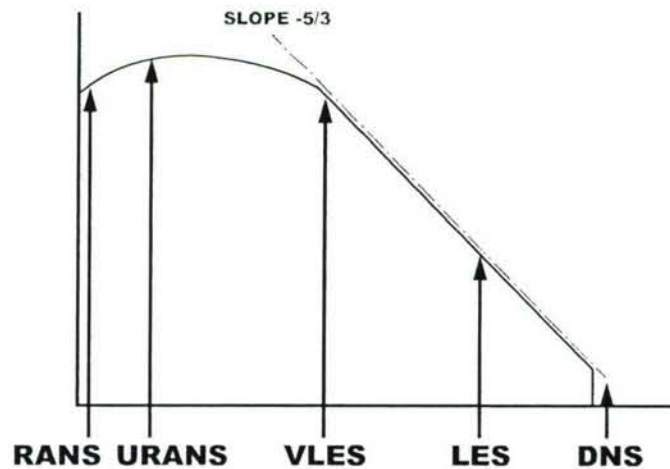


Figure 1. Illustration of turbulence modeling (RANS-DNS) on a 3D energy spectra.

Classic LES subgrid-scale models (like Smagoriski or its dynamic model variants) are algebraic in nature and assume that equilibrium is present between the unresolved and resolved scales. Unfortunately, this equilibrium is actually present only on average. The motivation for using a transport equation model is both theoretical and practical. Theoretically, a transport equation model can capture the inherently non-equilibrium subgrid-scale turbulence physics more accurately. Practically, a transport equation based model can easily transition to a RANS or URANS model at coarse mesh resolutions. In low Mach number flows, the solution for the pressure dominates the calculation time, and the solution of additional transport equations are not expected to incur a large performance penalty. In high Mach number flows, the turbulence equations can be advanced less frequently and with a larger timestep, and the conclusion remains the same.

Classical algebraic LES models require the mesh spacing to lie in the self-similar inertial range of the energy spectrum. In practical applied simulations is not always possible to ensure that the mesh spacing lies in some inertial or self-similar regime. Near walls, there is no inertial range. In

complex flows predicting the inertial range a priori is practically impossible. A model that can perform at any mesh resolution will not be restricted by this current LES limitation on the mesh size. In addition, in many situations it is wasteful to perform LES in regions of the computation (flat plate boundary layers and simple mixing layers) where RANS models are known to perform quite well. LES becomes more useful if the model can function in the RANS limit for those regions of the flow that are not too complex. Classic LES models cannot function in the RANS (coarse mesh) or DNS (very fine mesh) limits because the mesh size is no longer a useful indicator of the turbulent length scales (as it is in the inertial range).

There are a number of existing hybrid turbulence models that are designed to be able to address these issues and perform over a broad range of the mesh spectrum (i.e. URANS down through LES). For example, the very popular DES (detached eddy simulation) model¹⁹ behaves like a RANS or URANS model near walls, but away from walls the lengthscale is changed to the mesh size and the model has an LES character. Other hybrid models do not change their character based on location relative to a wall, but on whether the mesh is much smaller than the energy containing turbulence scales (leading to LES) or not (leading to RANS or URANS). The earliest implementation of such an approach Speziale²⁰ used classic LES (Smagorinsky) and RANS (k/ϵ) models to solve for both an LES and a RANS eddy viscosity and then blended these two viscosities together based on a function of the mesh size. Girimaji¹⁰ has developed a hybrid model (PANS) that can change its character based on input from the user (the user sets the desired ratio of modeled turbulent kinetic energy). The results presented in the following report can not be obtained with any of these previous approaches.

In this research, a self-adapting turbulence modeling approach was developed that works for any mesh resolution and over the entire energy spectrum. It can therefore do, RANS, URANS, VLES, LES and even DNS. More importantly, the character of the model is not set by the user (like PANS) or geometric location (like DES), but instead will adapt to whatever level the mesh can support. The proposed approach therefore models only as much kinetic energy is necessary (for a given mesh) and resolves as much of the energy using first principals as possible. It is not technically correct to consider the proposed approach to be a hybrid model in the classic sense (though it has many similarities to those models) because it does not blend an LES and a RANS model together. The proposed approach is a ‘universal’ modeling approach. It is a single model that works naturally at any mesh resolution.

2. A Universal Model

The classic mathematical theory behind RANS and LES makes these two modeling approaches look fundamentally different. RANS is based on ensemble averages and LES on filtering. At first glance, the possibility of a single model that does both (without some sort of switch or blending function) seems remote. However, a closer examination by Germano⁸ revealed some very important insights. Most importantly, the exact but unclosed governing equations for RANS and LES (and URANS, VLES and DNS) are all mathematically identical. While the RANS equations can be derived from the assumption of ensemble averaging and the LES equations from filtering operations, these assumptions are overly restrictive and neither system *must* be derived with those assumptions. The only required assumption is that the velocity field can be split into two parts and that this splitting operation commutes with differentiation. With this assumption the equations for turbulence evolution are (from Appendix A),

$$\frac{\partial \bar{u}_i}{\partial t} + \frac{\partial}{\partial x_j} (\bar{u}_i \bar{u}_j) = -\frac{\partial \bar{p}}{\partial x_i} + \nu \frac{\partial^2 \bar{u}_i}{\partial x_j^2} - \frac{\partial R_{ij}}{\partial x_j} \quad (1)$$

where \bar{u}_i and \bar{p} are the computed velocity and pressure and $R_{ij} = \overline{u_i u_j} - \bar{u}_i \bar{u}_j$ is the unknown turbulent stress tensor. The exact (but unclosed) evolution equation for this stress tensor is,

$$\begin{aligned} \frac{\partial R_{ij}}{\partial t} + \bar{u}_k \frac{\partial R_{ij}}{\partial x_k} = & \nu \frac{\partial^2 R_{ij}}{\partial x_k^2} - (R_{jk} \frac{\partial \bar{u}_i}{\partial x_k} + R_{ik} \frac{\partial \bar{u}_j}{\partial x_k}) - \frac{\partial}{\partial x_k} T_{ijk} \\ & - (\langle \frac{\partial p}{\partial x_i}, u_j \rangle + \langle \frac{\partial p}{\partial x_j}, u_i \rangle) - 2\nu \langle \frac{\partial u_i}{\partial x_k}, \frac{\partial u_j}{\partial x_k} \rangle \end{aligned} \quad (2)$$

where the bracket operation is given by $\langle a_i, b_j \rangle \equiv \overline{a_i b_j} - \bar{a}_i \bar{b}_j$ and the turbulent transport is defined by $T_{ijk} \equiv \overline{u_i u_j u_k} - \bar{u}_i \bar{u}_j \bar{u}_k - \bar{u}_i R_{jk} - \bar{u}_j R_{ik} - \bar{u}_k R_{ij} - \bar{u}_i \bar{u}_j \bar{u}_k$. Note that the stress tensor can also be defined using the bracket notation, $R_{ij} = \langle u_i, u_j \rangle$. The turbulent transport and bracketed terms require a model if the system is to be solved. In RANS the overbar might denote an ensemble average. In LES the overbar might be an explicit filtering operation. However, it can also be an implicit operation, because in practice when these equations are modeled and then solved on a computer, the overbar operation is never actually performed. In this case, it is assumed that an overbar represents whatever the calculation computes. It is not possible to prove that an implicit filter commutes with differentiation but it is a fairly reasonable assumption to make at least to first order.

To accurately model some of the terms in (1) and (2), a third 'scale' equation is often postulated that captures the turbulent energetic length or timescale. The epsilon equation¹² is perhaps the most commonly used scale equation, but omega (inverse timescale)²² and lengthscale equations^{15,18} also are possible and can be advantageous. In theory, these scale equations can be derived from first principals but the number of modeling assumptions, in practice, makes them largely empirical.

Starting from these exact equations numerous modeling approaches are possible. In order of increasing simplicity a brief description is given for some of the more common approaches. Reynolds stress transport (RST) models use a modeled form of the tensor equation (2). The more popular two-equation RANS models (such as k/ϵ or k/ω) are a simplification of equation (2) from a tensor equation to a scalar equation (by taking its trace). The primary unknown must then be reconstructed from this scalar kinetic energy, k , using a hypothesized algebraic relation such as the eddy viscosity hypothesis. One-equation models, such as the Spalart-Allmaras¹⁹ model used as the basis for DES²⁰ solve a single transport equation directly for the eddy viscosity and use an algebraic expression for the lengthscale (such as the distance to the wall). Classic LES models (such as Smagorinsky and its variants) are the simplest models of all, they solve no transport equations for the turbulence variables but algebraically obtain the necessary length and timescales from the mesh size and the resolved velocity gradients.

Traditionally, LES models have used the very simplest modeling approach because of the issue of cost. Early attempts at using more complex transport equations were deemed not worth the extra effort. Deardorff⁷ used a RST model and Schumann¹⁷ used a k/ε model. Models and computing power have changed considerably in the 30 years since those first simulations. Some more recent LES models now carry a single transport equation. This is certainly the case in DES, and a kinetic energy transport equation was used by Ghosal et al.⁹. As the mathematical analysis of Germano⁸ makes clear, there is no fundamental reason why these more complex modeling approaches (used currently only by RANS models) cannot also be applied to LES. The apparent natural evolution of turbulence models to include more physics and therefore more complexity, suggests that two-equation and RST transport models for LES are, in fact, the next logical step. The ‘universal’ k/ε model and its results are presented first (section 3), and the RST model and its performance are present in section 7.

3. A Two-Equation LES Model

The classical k/ε equation system is used in this first section in order to reach the largest audience possible. There are very good reasons to prefer other two-equation model systems. However, since all two-equation transport models are closely related, the proposed modeling ideas can be easily generalized to these other transport equation models. It should be very easy to implement the model presented in this section into existing commercial and industrial CFD codes.

The unclosed equations (1) and (2) will be modeled using the following transport equations,

$$\frac{\partial u_i}{\partial t} + \frac{\partial}{\partial x_j} (u_i u_j) = -\frac{\partial (p + \frac{2}{3}k)}{\partial x_i} + \frac{\partial}{\partial x_j} [(v + v_T \alpha) (\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i})] \quad (3)$$

$$\frac{\partial k}{\partial t} + \frac{\partial}{\partial x_j} (k u_j) = \frac{\partial}{\partial x_j} [(\frac{v + v_T}{\sigma_k}) \frac{\partial k}{\partial x_j}] + \alpha P - \varepsilon \quad (4)$$

$$\frac{\partial \varepsilon}{\partial t} + \frac{\partial}{\partial x_j} (\varepsilon u_j) = \frac{\partial}{\partial x_j} [(\frac{v + v_T}{\sigma_\varepsilon}) \frac{\partial \varepsilon}{\partial x_j}] + \frac{\varepsilon}{k} [C_{\varepsilon 1} P - C_{\varepsilon 2} \varepsilon] \quad (5)$$

where the overbar on the velocity and pressure are now dropped for convenience. The production is given by $P = v_T (u_{i,j} + u_{j,i}) u_{i,j}$ and eddy viscosity is given by $v_T = C_\mu \frac{k^2}{\varepsilon} (\frac{k}{k + k_r})$.

It is now necessary to distinguish between the modeled (or unresolved) turbulent kinetic energy, k , and the resolved kinetic energy, $k_r = \frac{1}{2} (u_1 u_1 + u_2 u_2 + u_3 u_3)$, which is calculated from the resolved velocity.

The constants are fairly standard k/ε constants, $C_{\varepsilon 1} = 1.55$, $\sigma_\varepsilon = 1.2$, $\sigma_k = 1.0$, $C_\mu = 0.18$. The parameter $C_{\varepsilon 2} = \frac{11}{6} f + \frac{25}{Re_T} f^2$ is sensitive to the local unresolved turbulent Reynolds number

$Re_\tau = \frac{k^2}{\nu \epsilon}$ of the modeled turbulence via the function $f = \frac{Re_\tau}{30} (\sqrt{1 + \frac{60}{Re_\tau}} - 1)$ as per the analysis of Perot and de Bruyn Kops¹⁴. This varies C_{ϵ_2} from its theoretical limits of 11/6 at high Reynolds numbers to 3/2 at low Reynolds numbers. Any Reynolds number dependent C_{ϵ_2} would probably be sufficient. Reynolds number dependence is important in LES because the effective turbulent Reynolds number $Re_\tau = k^2/\nu \epsilon$ becomes small as the mesh is refined (and goes towards zero for DNS). For incompressible flow, the pressure in equation (3) is determined from the incompressibility constraint, $u_{j,j} = 0$.

The two major differences in this equation system from the classical k/ε model is the definition of the eddy viscosity which now contains the kinetic energy ratio, and the presence of the additional parameter α which is discussed in the next section.

3.1 Backscatter of Energy

One key component of a self-adaptive turbulence model is that it must be able to backscatter energy from the unresolved (modeled) turbulence to the calculated (resolved) velocity field. For example, a fine grid (128^3) simulation of isotropic turbulence would resolve most of the energy and very little would need to be modeled. If this simulation were set up incorrectly, and most of the energy were defined to be modeled and very little resolved (i.e. initialized from a RANS simulation), the adaptive turbulence model ought to remove energy from the modeled kinetic energy and energize the resolved velocity field to correct this error. Figure 2 illustrates the behavior of backscatter (and the physically more dominant - forward scatter) for a 1d energy spectra. Here the resolved turbulence is on the left and the modeled turbulence is to the right (shaded), the arrows indicate the direction of energy flow.

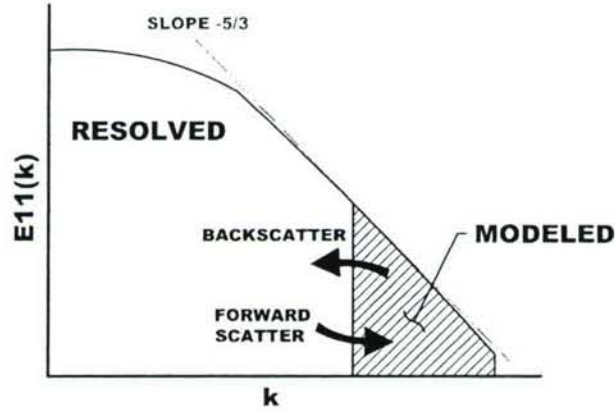


Figure 2. Illustration of backscatter and forward scatter on a 1D energy spectra.

With the ability to backscatter, a turbulence model can automatically correct errors in the initial conditions, and most importantly, perform model adaptation without user intervention. The idea of allowing backscatter in a turbulence model is not a new one. It has been shown by Chasnov³ and Carati¹ et al., that the $-5/3$ power law in isotropic decay is better predicted by LES (dynamic) models that account for backscatter. Along similar lines, classical DES can not backscatter energy but it has been recently shown by Piomelli et al.¹⁶ that adding noise, a crude form of energy backscatter, to the DES model improved results for channel flow.

If $\alpha = 1$, the proposed system (Equations (3), (4), (5)) is very close to a classic k/ε model. The proposed formulation assumes that the turbulent stress tensor is reconstructed using the eddy viscosity hypothesis, $R_{ij} = \frac{2}{3} k \delta_{ij} - \nu_T \alpha (u_{i,j} + u_{j,i})$. This of course is the simplest reconstruction hypothesis to use. Nonlinear eddy viscosity and algebraic Reynolds stress models use more complex (but still algebraic) reconstructions. Again, the proposed ideas can easily be generalized to that context as well. The simplest model possible is used in this work in order to focus as directly as possible on the key idea - that it is possible to develop turbulence models that automatically work at any mesh resolution.

RST models based on equation (2) can, and do, backscatter energy. We will show this effect later. But RST models are more complex and less familiar, so we wish to enable the two-equation models to perform as universal models, and this will require adding backscatter. The classical two-equation models are based on a positive definite eddy viscosity which is too simplified an assumption (eddy viscosity should actually only be positive on average) and therefore classical two-equation models can not backscatter energy. The eddy viscosity is always positive and therefore the model always removes energy from the mean flow and puts it into the modeled kinetic energy, k , where it is eventually dissipated by ε to heat. It can never move energy from the model to the resolved scales. This is the correct average behavior for a turbulence model, but not necessarily locally correct (in space or time).

The additional parameter α has been added to the classic k/ε model to correct this important flaw and control the energy flow. Usually α is positive (and order 1), but it can become small or even negative. When $\alpha < 0$ the model is backscattering energy. This parameter is not a model constant, instead it is a field that varies in space and time, so that backscatter can happen in different regions at different times. The transfer variable, α , also appears in Equation (4), the k -equation, so that the total kinetic energy (modeled k plus resolved k_r) is a conserved quantity and can only disappear via dissipation to heat. Its presence is not necessary in the scale equation, Equation (4), so we do not include it.

When $\alpha < 0$, the eddy viscosity in (4) is essentially negative. Negative viscosity is anti-diffusive, it amplifies (rather than damps) existing resolved velocity fluctuations. It amplifies small wavelength modes (those closest to the mesh resolution) the most rapidly. This is a very reasonable model for backscatter. It is not injecting energy via some random forcing of the resolved flow, rather it works to enhance the existing instabilities and modes. Moreover, the energy transfer is local in spectral space. It tends to take energy from the model (which has most of its energy at scales just below the mesh resolution) and preferentially delivers it to the resolved flow at almost the same length scale (but just above the mesh resolution).

The model for α the energy flow parameter, uses ideas from error estimation for mesh adaptation. If the error in the computed solution is small, then the mesh is deemed to be sufficient for first principals simulation and the model should go away (the modeled kinetic energy should become very small). This is akin to the situation described earlier, where a 128^3 grid is initialized with a RANS solution and some small resolved flow fluctuations. The model will detect this situation as very highly resolved (lots of mesh resolution). Due to energy conservation inherent in Equations (3) and (4), the modeled kinetic energy can not just disappear. In order to become small the modeled kinetic energy must be transferred somewhere - the only possibility is to the resolved flow. Small estimated errors (in the resolved quantities) implies there should be energy backscatter (the mesh can handle more fluctuations via direct simulation). So small errors result in $\alpha < 0$.

At some point later in time, the resolved velocity on the 128^3 mesh will be fully energized. If the Reynolds number is high, then a 128^3 mesh is still not sufficient to perform DNS (a model is still necessary). In this case the error estimate will become larger and larger until α becomes positive and a normal forward energy scatter down the energy cascade will be imposed. The larger the error, the more energy is fed to the modeled kinetic energy and the more the model influences the resolved flow evolution. If the Reynolds number is low enough, then a 128^3 mesh may actually be sufficient resolution for DNS and the error estimate stays small. In this case the model continues to backscatter until the model has almost no energy. At this point the model has no affect on the resolved modes and DNS is achieved.

3.2 Energy Transfer Variable

The proposed equation for controlling the energy transfer is,

$$\alpha = 1.5(1.0 - C^* (\frac{k}{k + k_r})^2 [(\frac{\Delta x_i}{\sqrt{k_r}} \frac{\partial \sqrt{k_r}}{\partial x_i})^2 + 0.11]^{-1}) \quad (6)$$

where k_r is the resolved kinetic energy (at a certain location and time), k is the modeled kinetic energy, and $C^* = 0.28$. The quantity $(\Delta x_i \frac{\partial \sqrt{k_r}}{\partial x_i})^2 / k_r = ((\Delta x \frac{\partial \sqrt{k_r}}{\partial x})^2 + (\Delta y \frac{\partial \sqrt{k_r}}{\partial y})^2 + (\Delta z \frac{\partial \sqrt{k_r}}{\partial z})^2) / k_r$ is a dimensionless measure of the error (similar to what is sometimes used in mesh adaptation). In this formulation the resolved kinetic energy $k_r = \frac{1}{2}(u_1^2 + u_2^2 + u_3^2)$ is the indicator function that is being used to estimate the mesh resolution. If the flow is DNS or over-resolved (like a RANS initial condition on an LES mesh) then this quantity is small, its inverse is large (but limited away from infinity by the fairly arbitrary 0.11 term), and the model tends to backscatter energy. In contrast, normal energy transfer (from resolved scales to the modeled scales) occurs in the regions of the flow where the gradient length scales are comparable to the mesh size. On very coarse meshes, RANS like behavior should be recovered. In this limit, k_r is expected to be very small, but note that

$(\Delta x_i \frac{\partial \sqrt{k_r}}{\partial x_i})^2 / k_r$ will remain finite and independent of the magnitude of the resolved fluctuations.

For the simulations of isotropic turbulence performed, this term obtains an average value of 0.8 in the RANS limit. This means that $\alpha \rightarrow 1.5 - 0.42/(0.8 + 0.11) \approx 1.04$ and the standard RANS model is very closely recovered in the RANS limit.

The particular form of the energy transfer function was developed largely by intuition and tuned solely to obtain the correct limits. Many other functional expressions and/or indicator quantities are certainly possible. The goal of this work is not to advocate for this particular function but to demonstrate that self-adaptive turbulence models are possible, and this particular function serves this purpose adequately.

3.3 Numerical Method

Simulations of isotropic turbulence are frequently performed with Fourier spectral methods that use discrete FFTs to solve the pressure Poisson equation. Spectral methods are not indicative of what is used in commercial codes. On the other hand, commercial codes tend to have excessive numerical dissipation (because high robustness is sought). This numerical dissipation can lead to poor results – even in the DNS cases that do not involve the model. This work compromised between the two extremes by using a 2nd order Cartesian staggered mesh method¹¹ (see Figure 3) with exact projection² for the pressure solution. This is essentially a low order finite volume method (like the commercial codes) that conserves mass, energy, and vorticity properly (like the spectral methods). The code is fully parallel (using MPI) and optimized for execution on PC clusters.

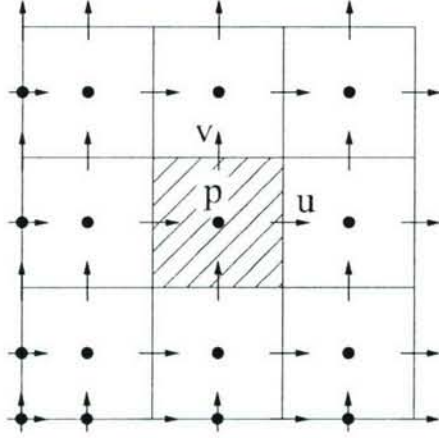


Figure 3. Fully staggered arrangement of velocity (u,v) and pressure (p). The actual simulation is three dimensional.

4. Isotropic Decaying Turbulence

We wish to demonstrate the validity of the approach using one of the simplest turbulence models (k/ϵ) and one of the simplest turbulent flows possible (decaying turbulence). This is intentionally done so there is no room for ambiguity. Neither the complexity of the problem nor the model can obfuscate the results.

It should be noted that other hybrid methods, such as DES, do not work well for this very simple problem. In the absence of walls DES always does LES, and when the mesh becomes too coarse it will fail like any other classical LES model. PANS will not adapt as the turbulence decays, so when the turbulence has decayed into the DNS limit, PANS will still be applying an LES eddy viscosity.

Isotropic decaying turbulence was calculated using periodic boundary conditions on a box of size of $18\pi \times 18\pi \times 36\pi$. This initial box size was used in order to compare our results with an independently performed DNS simulation of this flow. Our mesh size will always be double in the third direction so that the actual mesh spacing is always equal in all three directions.

Figure 4 shows a mid-plane slice of an initial condition $256 \times 256 \times 512$ isotropic field for reader visualization. It is apparent from figure 4 that the $256 \times 256 \times 512$ initial condition is turbulent and isotropic (no preferential direction). While the following results focus on isotropic turbulence because of its simplicity, it must be stressed that this methodology can easily be applied to any flow field, and makes no assumptions about the problem being solved.

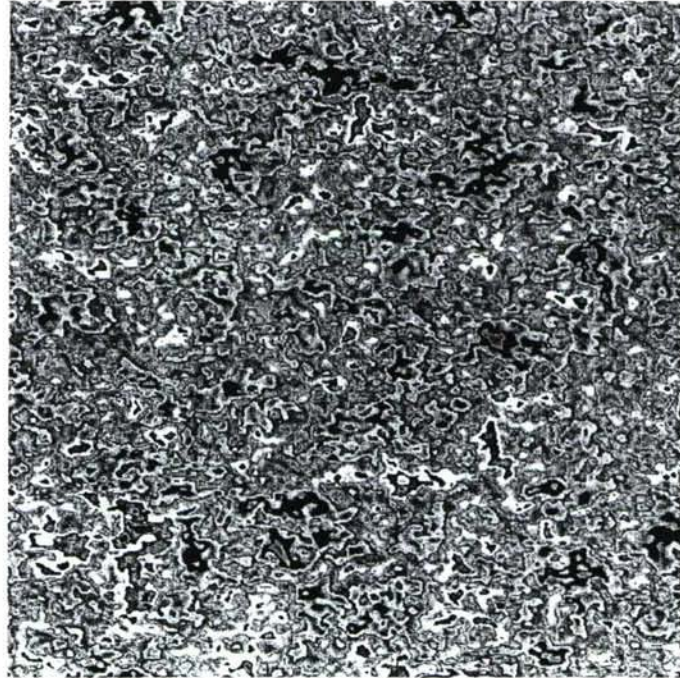


Figure 4: Mid-plane slice (x-y plane) through initial $256 \times 256 \times 512$ isotropic field, $Re = 640$.

4. 1 High Re Isotropic Decay

In isotropic turbulence, the turbulent kinetic energy decays with time. This process can be simulated with a simple RANS model using only one spatial grid cell, with LES using many cells, or with DNS using enough cells to resolve the smallest length scales of motion. Results from such a test are shown in Figure 5. The ordinate shows total kinetic energy ($k + k_r$) normalized by the initial total kinetic energy at time $t=0.0$, the abscissa τ , is given by simulation time (seconds) non-dimensionalized by the inverse time scale ($\frac{\epsilon_t}{k_r}$) at time $t=0.0$ (where ϵ_t indicates total quantities). Normalization of the abscissa can be interpreted as 'eddy' turn over times.

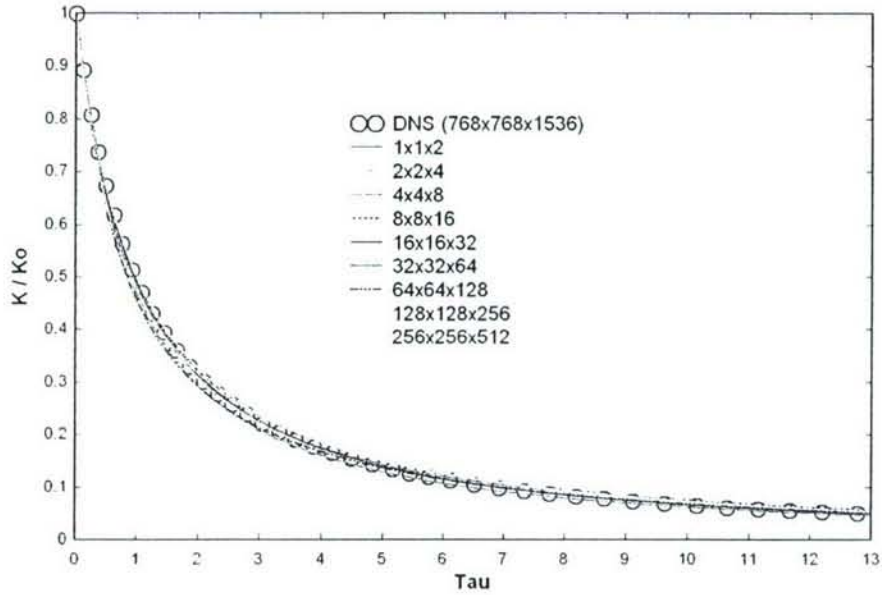


Figure 5. Total kinetic energy predictions of isotropic decay at initial $Re=640$.

The initial turbulent Reynolds number, $Re_\tau = \frac{k_r^2}{\nu \epsilon_t}$, for this test case is 640, comparable to the

Comte-Bellot and Corrsin 1971 experiment⁴. The DNS result was performed independently by de Bruyn Kops⁵ on a $768 \times 768 \times 1536$ mesh using a Fourier spectral method and is given by the large circles. Many aspects of the DNS (including spectra) have been closely compared with Comte-Bellot and Corrsin and shown⁶ to agree well. The mesh resolution is identical in each direction, with the box size in the z -direction twice as large.

A number of different simulations were performed using mesh resolutions from $1 \times 1 \times 2$ to $256 \times 256 \times 512$. In each case the model is identical and only the mesh and initial conditions are changed. The initial conditions are obtained from the full DNS. The DNS initial condition is not random Fourier modes - but full Navier-Stokes turbulence that was obtained by running for a long time with as little forcing of the large scales as possible, to maintain the kinetic energy.

Each model initial condition was formed by averaging the same initial $768 \times 768 \times 1536$ velocity field to the appropriate mesh size using a simple box average. The initial modeled kinetic energy at each mesh location was then determined by comparing the exact $768 \times 768 \times 1536$ velocity field to the smoother coarse mesh field and calculating the sum of its difference. The modeled dissipation was calculated similarly, by using a box average of the exact DNS dissipation field, and comparing to the known total dissipation at time zero. Because dissipation tends to occur at the smallest scales, the modeled dissipation is very close to the total dissipation (very little dissipation is resolved) except for the very finest mesh ($256 \times 256 \times 512$).

The results in Figure 5 are boring to look at but profound. They show that at any mesh resolution, the model predicts the decay of the turbulence accurately. The very smallest mesh resolution is clearly a RANS simulation and the largest mesh, $256 \times 256 \times 512$, is an LES (bordering on DNS) simulation. The intermediate resolutions might be considered URANS, VLES, or LES. The spectra for the initial conditions are shown in Figure 6. These spectra represent VLES ($32 \times 32 \times 64$), LES ($64 \times 64 \times 128$, $128 \times 128 \times 256$), and LES/DNS ($256 \times 256 \times 512$) simulations. The box average of the initial data takes some energy from the lowest wavenumbers but most of the energy from the highest wavenumbers. All energy that is not resolved is modeled by the spatially and temporally varying variable k .

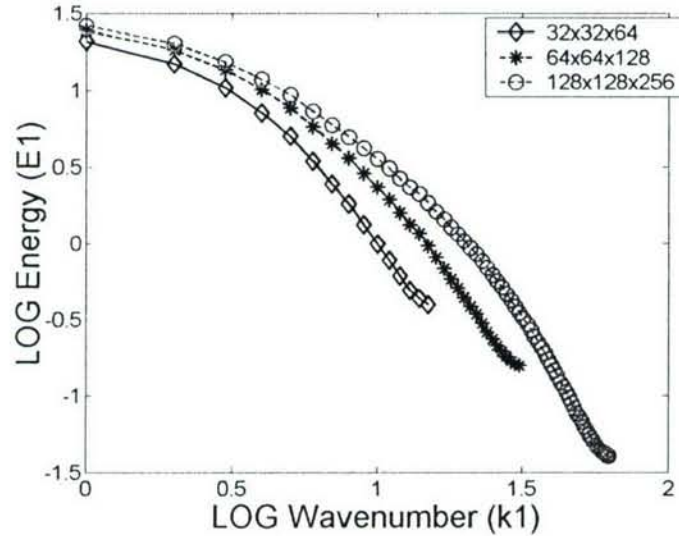


Figure 6. Spectrum of initial conditions ($32 \times 32 \times 64$, $64 \times 64 \times 128$, $128 \times 128 \times 256$, $256 \times 256 \times 512$) for ($Re=640$) isotropic decaying turbulence.

4.2 Kinetic Energy Ratio

It has been hypothesized that RANS equations must give RANS (essentially the 1x1x2) results irrespective of the mesh resolution used to solve those equations. This would indeed be the case, if the equations were linear. Here we clearly show that in fact, different meshes produces different solutions to this coupled equation system (even though the total behavior, figure 5, remains the same for all meshes).

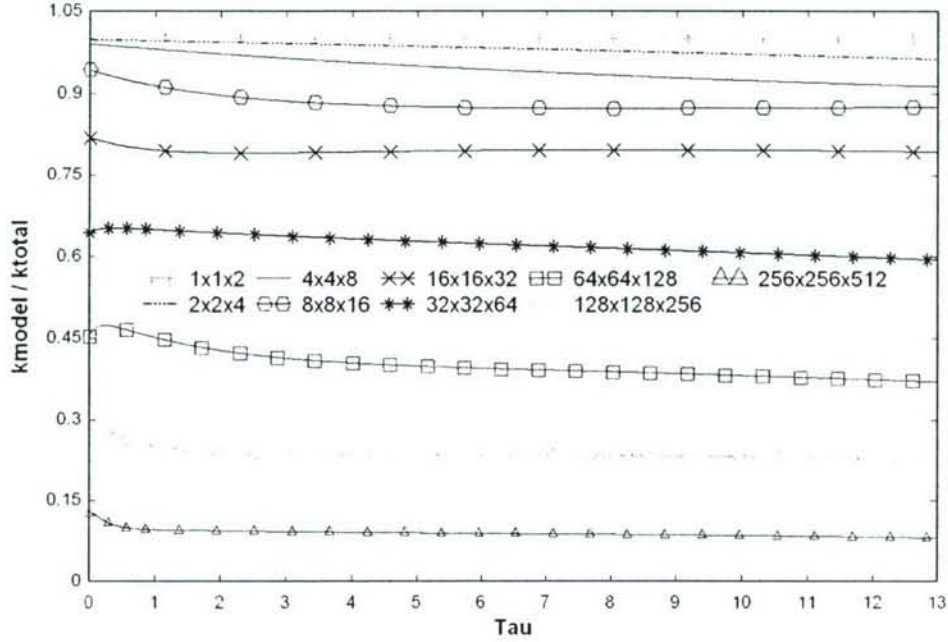


Figure 7. Ratio of modeled kinetic energy to total kinetic energy, $Re=640$.

The ratio of the modeled kinetic energy to the total kinetic energy is shown in Figure 7 with one curve for each of the mesh resolutions. The 1x1x2 solution is the top curve, with all its energy contained in the model (giving a ratio of 1.0) and the 256x256x512 simulation is the bottom line, with the smallest ratio of modeled kinetic energy (<10 percent). Note that these curves are relatively constant and decrease slightly with time as the simulation proceeds. They do not approach each other, or the RANS limit (1.0). Each mesh resolution is operating with a different average value for the modeled kinetic energy. Those resolutions that require less modeling of the energy start with less model energy and maintain that lower level.

The LES solutions stay entirely unsteady and three-dimensional. The slight decrease in the ratio over time is the correct behavior. It is due to the fact that over time the Reynolds number of the flow is slowly decreasing and the mesh can, and therefore does, resolve a larger percentage of the turbulent fluctuations. If the simulation was run long enough, then the Reynolds number would drop sufficiently for the 256x256x512 mesh to perform DNS, and at this point the ratio should be essentially zero. The small variation at the beginning of each simulation shows the initial

condition is close to the correct ratio supportable by that mesh, but not perfect. The long time behavior of this ratio is shown in Figure 8. These decay rates are consistent with the Reynolds number decay rate ($Re \sim t^{-1/5}$).

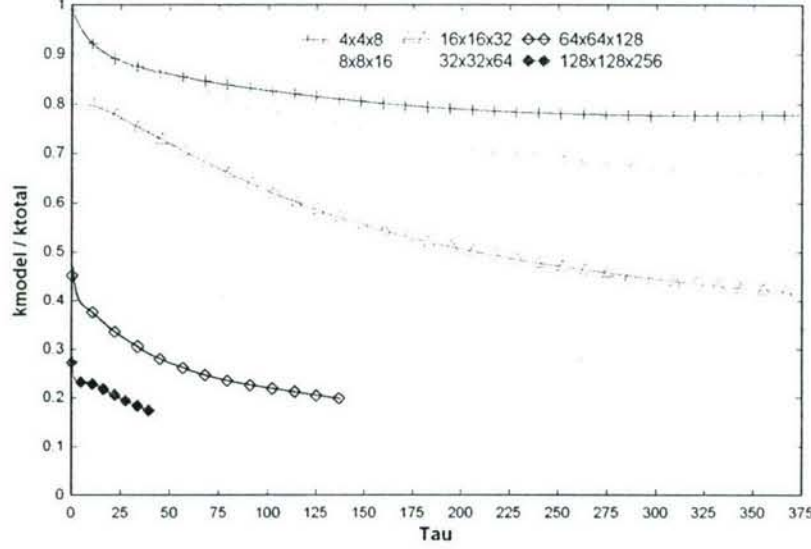


Figure 8. Ratio of modeled kinetic energy to total kinetic energy for extended time, $Re=640$.

Figure 8 shows that the coarse mesh (4x4x8) might approaching the 'RANS' solution (a ratio of 1.0) at very long times. When the mesh is larger than the largest eddies this is possible. The finer meshes (8x8x16 - 128x128x256) have enough resolution to compute a solution that does not return to the RANS limit. It is hypothesized that steady (RANS-like) solutions occur when the resolution is not sufficient to maintain an energy cascade. For this test case, the initial large eddy lengthscale of the turbulence ($L = k^{3/2}/\epsilon$) is 6.0, and the 8x8x16 simulation has a mesh size close to 7.0. The 4x4x8 simulation therefore has a mesh size that is twice the size of the energy containing eddies. It can not resolve many eddies, and can not set up a nonlinear energy cascade. Nikitin et al.¹³ has noted that a similar affect occurs in very under-resolved DES simulations.

4.3 Perturbation of Initial Conditions

A truly adaptive model should be able to obtain the correct behavior from incorrect initial conditions. For example, it is of considerable interest to see if a 64x64x128 mesh initialized with a RANS solution can, over time, develop into a full LES simulation. In order to test the model in this way, the initial conditions were either smoothed or sharpened using a filtering operation. The filter used to alter the initial conditions was a nearest neighbor averaging procedure,

$$u_{ijk}^{filtered} = \beta u_{ijk} + (1 - \beta)(u_{i+1,j,k} + u_{i-1,j,k} + u_{i,j+1,k} + u_{i,j-1,k} + u_{i,j,k+1} + u_{i,j,k-1}) \frac{1}{6}. \text{ For smoothing, } \beta = 0 \text{ was}$$

used. This replaces the value at a mesh point by the average of its nearest neighbors. This type of filter removes energy primarily from the highly oscillatory modes with wavelengths close to the mesh size. In spectral terms it damps the spectra in the region just above the cutoff wave number.

The affect is shown in Figure 9, which shows the original initial spectra for the 64x64x128 simulation, and the spectra for the smoothed and sharpened initial conditions. Sharpening the velocity field is performed by using $\beta = 1.5$. This adds energy to the existing high frequency modes.

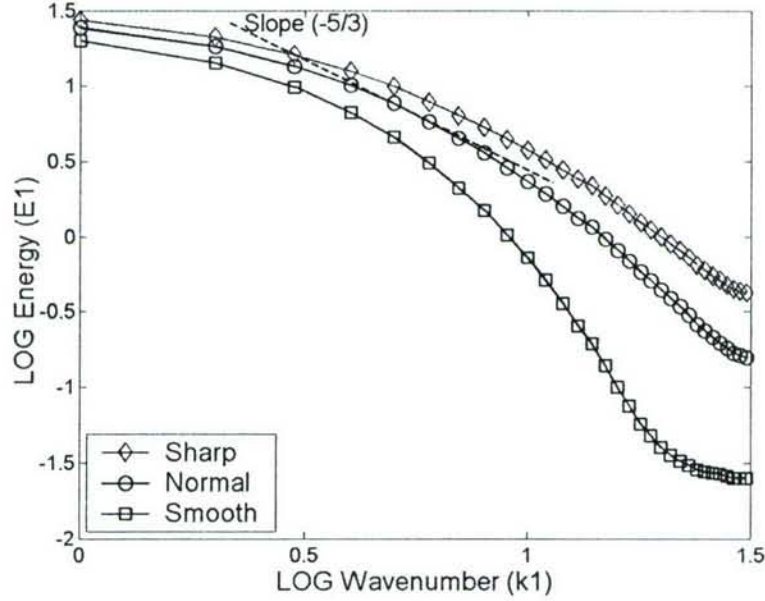


Figure 9. 1D energy spectra for 64x64x128 case, Re=640 (Sharp, Normal, Smooth).

Figure 10 shows the affect of smoothing and sharpening the initial conditions on the kinetic energy ratio. When smoothing is used, energy is removed from the resolved modes. In order to keep the total kinetic energy the same, the model now must start with more modeled kinetic energy. The ratio therefore starts higher than before. We would like this model energy to backscatter into the resolved velocity field, since the mesh can in fact support more turbulence than was introduced at the initial condition.

Note that as time proceeds the model achieves the same ratio irrespective of the initial conditions. At early times, the smoothed solution has less error and therefore backscatters somewhat more than the unperturbed initial condition. This removes energy from the model and makes the ratio decrease faster, so that it approaches its original state. A similar (but opposite) process happens when the spectra is sharpened. In this case, the model senses that the mesh can not support the input resolved fluctuations, and quickly sends that energy to the model (by damping the resolved velocities). Note that the rate at which the model adjusts to perturbed initial conditions depends on the mesh resolution. The higher mesh resolutions adjust more rapidly. It is hypothesized that the time it takes to transfer the energy scales on the timescale of the turbulence at the cutoff (transfer) lengthscale (k/ϵ).

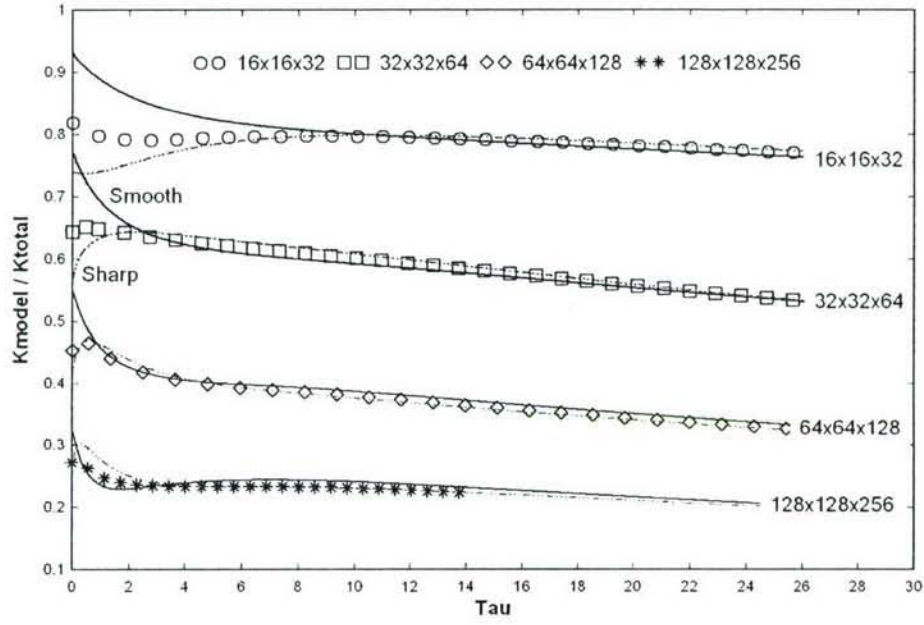


Figure 10. Ratio of modeled kinetic energy to total kinetic energy, $Re=640$, for perturbed initial conditions.

4.4 Low Re Isotropic Decay

In order to verify that the model works in the DNS regime, a lower Reynolds number simulation was performed which is well resolved (DNS) at the largest simulation of $256 \times 256 \times 512$. The initial condition for this lower Reynolds number case was generated by running the high Re simulation with a larger viscosity ($\nu = 0.3743 \text{ cm}^2/\text{s}$) and with a very small amount of modeled kinetic energy ($k = 10^{-3} \text{ cm}^2/\text{s}^2$). Using the higher viscosity reduces the Reynolds number and also quickly damps the smallest scales of the turbulence.

The turbulent time scale (k/ϵ_t), is plotted versus simulation time (solid line) in Figure 11 for the low Re case. As was expected, at early times the turbulence is adjusting and the curve is not linear. This is due to the simulation reacting to the increased viscosity and reduced modeled kinetic energy. Eventually, a power law decay for the kinetic energy is obtained, and the simulation is considered fully adjusted. A power law decay should appear as a straight line in Figure 11. A linear curve fit is given by the plus symbols. The inverse of the slope given by the curve fit is the kinetic energy decay exponent, $n \approx 1.38$. The simulation was considered to be well developed at a time of 0.14s. This field (at $t=0.14\text{s}$) was used in all the subsequent simulations, as the well developed low Reynolds number ($Re=211$) initial condition. The same box averaging procedure as the high Re number case was used to create initial conditions for the smaller meshes.

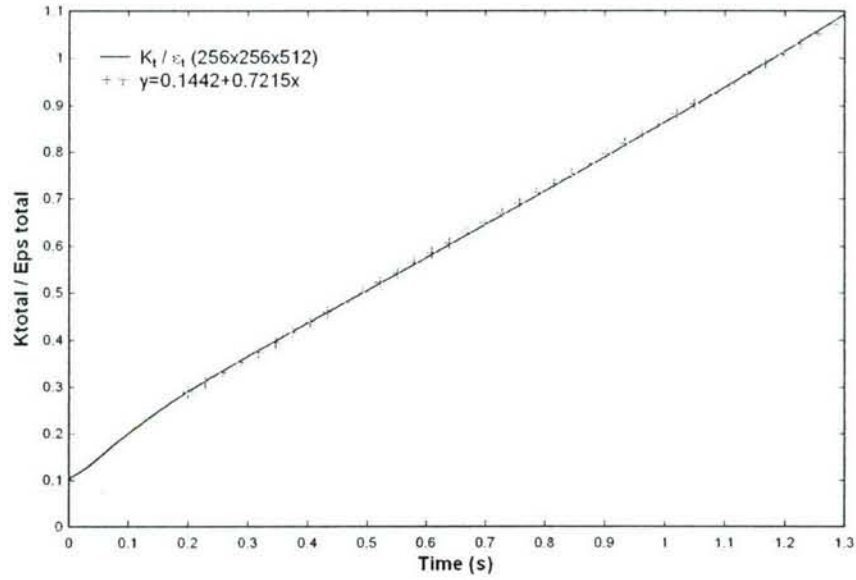


Figure 11. Turbulent time scale (k_t/ε_t) vs. simulation time for a 256x256x512 simulation with increased viscosity and small amount of modeled kinetic energy (DNS).

Figure 12 shows the total kinetic energy predictions for the low Re test case. The circles represent the essentially DNS simulation (though the model is on), the lines represent the various simulations from 1x1x2 through 128x128x256. These are in fairly good agreement with the DNS data, as was observed for the high Reynolds number case. The slight discrepancy at long times is most likely due to errors in the RANS model at lower Re.

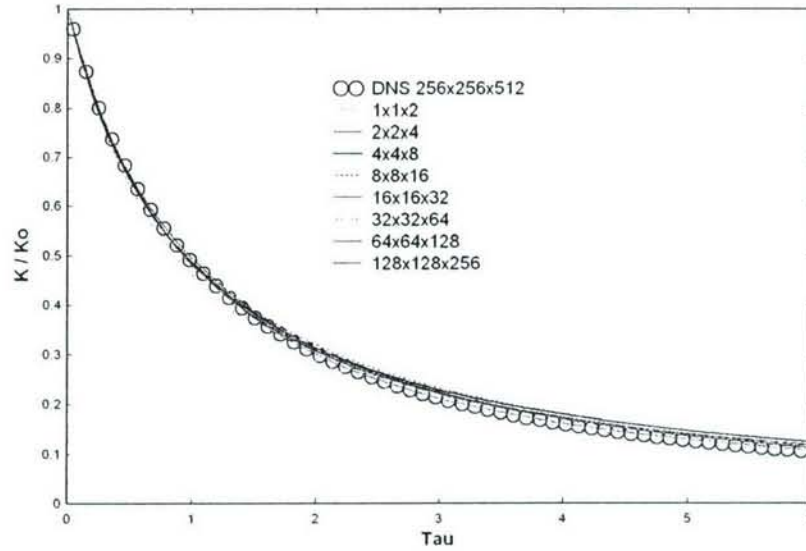


Figure 12. Total kinetic energy predictions of isotropic decay at initial Re=211.

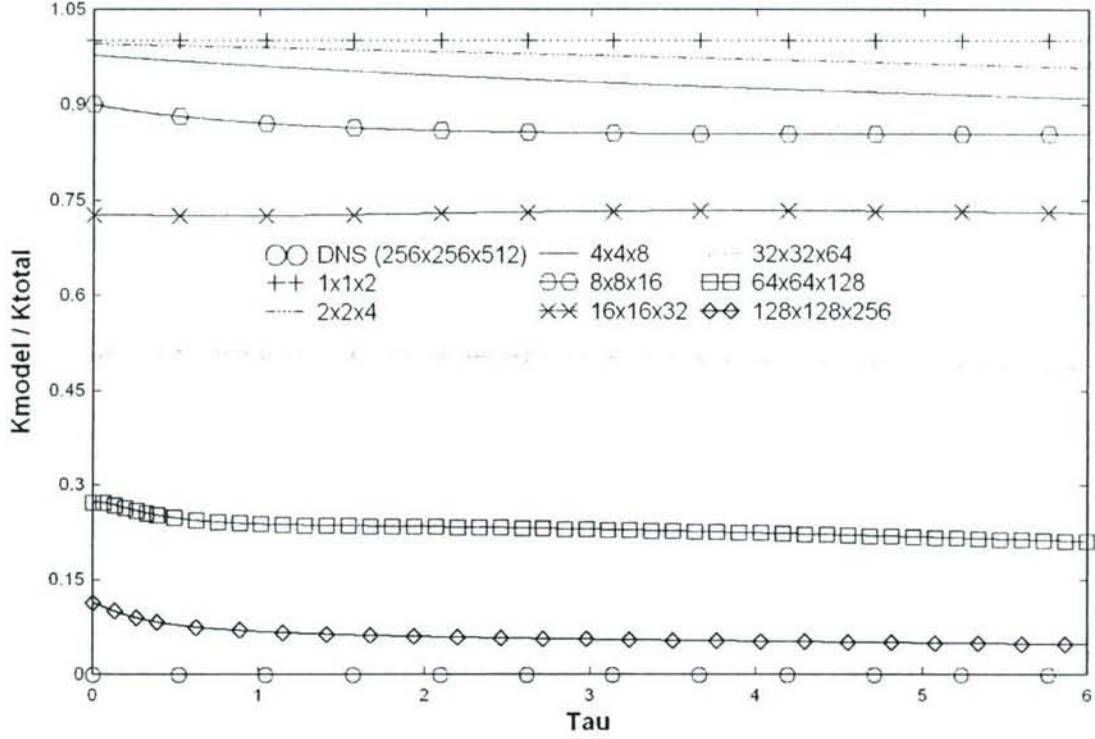


Figure 13. Ratio of modeled kinetic energy to total kinetic energy, $Re=211$.

The ratio of modeled kinetic energy is plotted in Figure 13. The full RANS simulation (1x1x2) is at the top of the figure (ratio of 1.0) and the DNS is shown at the bottom of the figure with a ratio of roughly 10^{-3} . Once again it is observed that there is no tendency for these solutions to move towards the RANS solution (ratio of 1.0). The LES solutions stay entirely unsteady and three-dimensional, and the correct decrease in this ratio over time is shown. The 128x128x256 mesh is close to DNS and the 256x256x512 mesh is clearly a DNS simulation even though the model is technically on.

4.5 Scaling

In classic LES models the lengthscale is assumed to be proportional to the mesh size, Δx , and it can be shown that the gradients scale like $u_{i,j} \approx \epsilon^{\frac{1}{3}} \Delta^{-\frac{2}{3}}$. The eddy viscosity is then constructed from this scaling and has the form, $\nu_T \approx u_{i,j} \Delta^2 \approx \epsilon^{\frac{1}{3}} \Delta^{\frac{4}{3}}$. The use of the mesh size to infer the lengthscale is why classic LES fails outside of the inertial range (close to the DNS or RANS regimes). It is also one reason why the proposed self-adaptive model (which predicts the lengthscale and does not infer it) can operate at any mesh size. However, the current model

should still obtain the classical LES scaling in the inertial subrange. In other words, the lengthscale predicted by the model ($L_m = k^{3/2}/\varepsilon$) should be proportional to the mesh size, when the model is operating in the LES regime.

Figure 14 looks at predicted lengthscale (in log scale) at a fixed time $t=0.5$ for the various meshes. At small values of Δx (large number of mesh points) it is obvious that the lengthscale is indeed a linear function of grid spacing as would be expected with classic LES. A reference line has been added (dotted line) with a slope of 1.0 for comparison. If a well defined $-5/3$ slope were to be observed in the energy spectra, one would assume that classic LES would closely match the reference line in that $-5/3$ region.

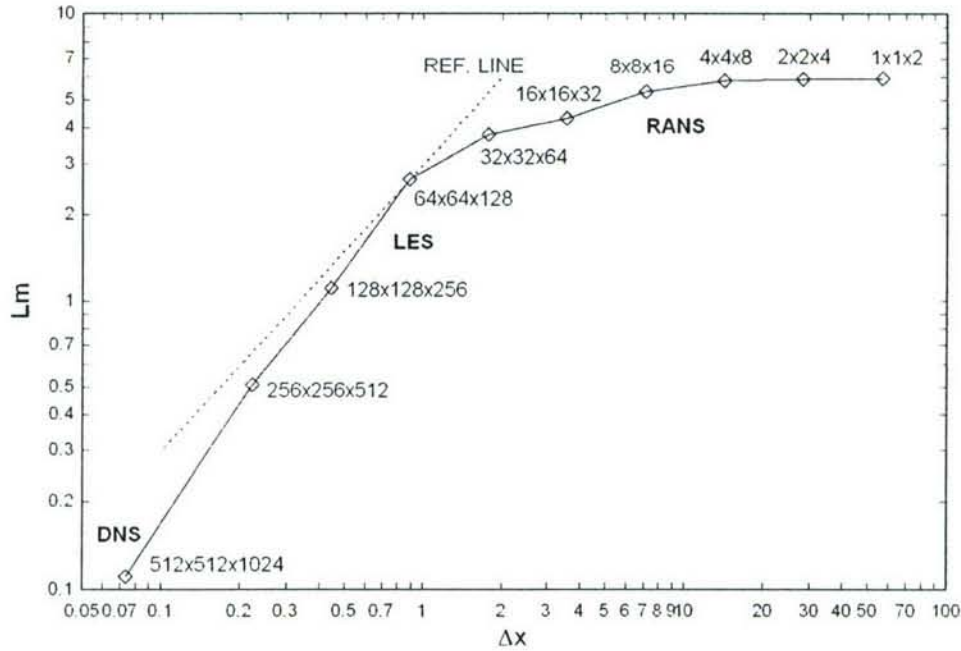


Figure 14. Lengthscale behavior at time = 0.5s.

Because the spectra at this moderate Reynolds number (Figures 6 and 9) show only a weakly observable $-5/3$ slope, it is not surprising that the lengthscale shown in figure 15 does not match the reference line exactly. One would also expect viscous effects (the influence of a viscous lengthscale) to reduce the scaling below linear and this is indeed observed. The figure clearly shows how the turbulent lengthscale is not related to the mesh spacing outside the LES regime.

Mesh resolutions smaller than $64 \times 64 \times 128$ do not exhibit the linear behavior because now the relevant turbulent lengthscale is no longer related to the mesh size. There is a transition region that one might call VLES ($32 \times 32 \times 64$) but not full LES. The $8 \times 8 \times 16$ mesh might be considered the start of URANS (as shown in Figure 14), where the largest scales can just be resolved, and unsteady 3D solutions maintained. These smallest mesh sizes are where the solutions can not

sustain a cascade because all the largest turbulence has scales that are smaller than the mesh size. Interestingly, Carati et. al.¹ conducted LES simulations for isotropic decaying turbulence and determined that the smallest mesh size possible for a classic (dynamic model) LES simulation was 48^3 . This is in very good agreement with Figure 14, and is just where the elbow in the curve occurs.

5 Parallel Programming

Even with current technological advances in computer hardware, large simulations can require an extraordinary amount of computer resources. Performing a $256 \times 256 \times 512$ simulation for these test cases requires the computer to solve approximately 34 million grid points. It is obvious that a standard PC will not be able to handle the computational requirements for this case anytime in the near future. In order to decrease simulation time and lower the memory requirements, the code was parallelized. This was accomplished by utilizing the Messaging Passing Interface (MPI) library. MPI is a library of functions for Fortran and C/C++ that distributes information from a single processor to multiple processors. Using more processors allows the larger simulations in this work to be accomplished.

5.1 Boundary Conditions

While MPI facilitates the transfer of data between processors, care must be taken to ensure the correct data is passed between neighboring processors. All simulations performed in this work use periodic boundary conditions as shown on the left side of Figure 15.

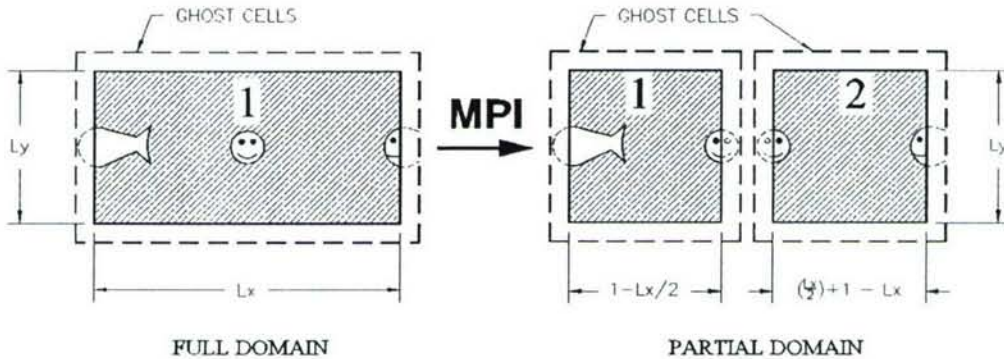


Figure 15. Single processor (left side) and MPI (right side) partitioning of the problem.

This figure (left side) shows how data is stored and distributed for periodic boundary conditions on a single processor. All of the interior points are within a domain of L_x and L_y (shaded area) and stored on a single processor. The ghost cells are used for the periodic boundary conditions and are given by the values of the interior points at each face in this rectangle. Here, Nemo the fish is seen swimming out of the left side of the domain. Because of periodicity, he is wrapped

around to the opposite face he just left, and is seen swimming into the domain from the right hand side. Similarly, if he swims out of any other face, he will wrap around to the opposite face he just left. This ensures that Nemo continues swimming inside the periodic domain.

Suppose that the same data set is desired to be split in half, then each processor should get half of the interior points as shown (right side of Figure 15). Here the x-direction, with an initial domain of (Lx) has been split in half $(Lx/2)$ for two processors. It is now apparent that care must be taken when determining the correct transfer of data between these two processors. The only way to correctly determine this information is by allowing every processor to locate its neighbor for each of the (cubic) domain's six faces. From Figure 15 for the partial domain, interior information must be shared with neighboring processors (given by the smiley face). However, at domain edges, the boundary information should wrap around to the opposing face, (given by Nemo). An algorithm was developed to do exactly this and is explained in detail in the next section.

5.2 Neighboring CPU Data

The algorithm used to compute neighboring CPU information is detailed in Appendix B. Figure 16 shows a parallel simulation consisting of eight processors. The left side of Figure 16 shows the data together, the right side shows the front and back plane separated for clarity with each of the 8 processors numbered from 0-7. Since each cube of data has six sides it is clear that there will be six neighboring CPU locations calculated for each processor. This algorithm uses simple integer division to allow each processor to find its correct neighbors for an arbitrary partition..

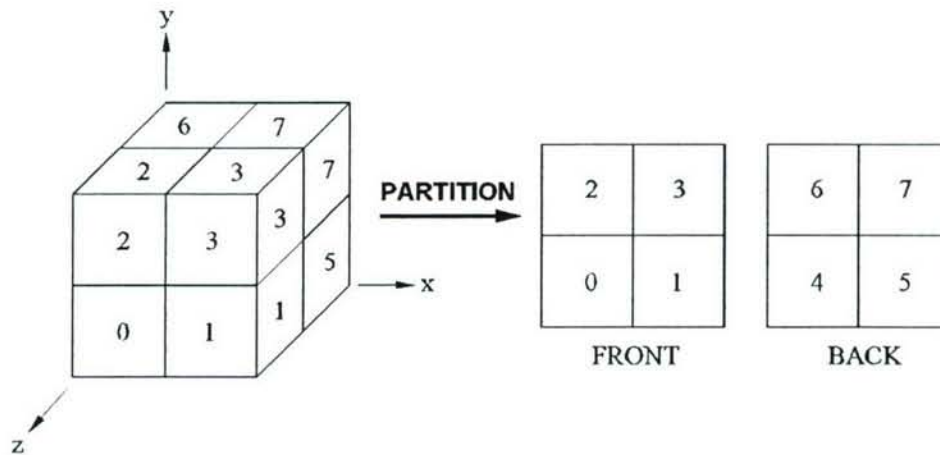


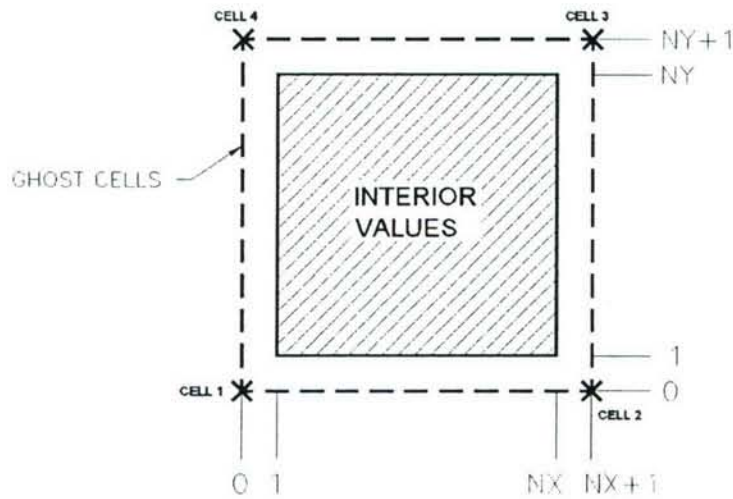
Figure 16. Neighboring CPU location.

By visual inspection the processor to the right of 0 is processor 1. There is no processor to the left of 0, however periodicity tells us that the information should wrap around from processor 1. The processor above 0 is processor 2, and below is also 2 (from periodicity). Finally, the processor in front of 0 is 4 (periodicity) and the processor behind 0 is also processor 4.

This simple reasoning is exactly what the algorithm computes in Appendix B. For the processor in question, the MPI assigned integer 'myid' is 0. Because there are two processors in every direction, $NXP = NYP = NZP = 2$. The values of ix, iy, and iz are calculated as 1. The CPU to the right of processor 0 will be calculated as (CPUinfo1 = 1), the CPU to the left (CPUinfo2 = 1), the CPU at the top and bottom are given by (CPUinfo3 = 2, CPUinfo4 = 2) respectively. Finally, the CPU located at the back and front of the CPU in question are (CPUinfo5 = 4, CPUinfo6 = 4) respectively. Simple integer division based on the variable 'myid' computes neighboring processor locations. The 'If' statements shown in Appendix B are used whenever a processor is on the edge of a domain (such as computing the processor to the left of 0). If any of these processors are on this domain edge, it is computed that boundary information will wrap around to ensure periodicity. It is clear from the above figure that the algorithm has correctly determined the location of the neighboring CPUs relative to processor 0.

5.3 Transfer of Boundary Condition Information

With everything in place, the actual passing of the boundary information will now be explained in detail. The cell indexing convention is shown in Figure 17. This shows that the interior points are given by the data values from 1-NX and 1-NY, where NX is the number of points in the x-direction and NY is the number of points in the y-direction. The ghost cells (dashed lines) used for the boundary conditions are placed at index locations around the interior values. The location of cell 1 (0,0), cell 2 (NX+1,0), cell 3 (NX+1,NY+1), and cell 4 (0,NY+1) are the four corners for this 2d example. The correct procedure must be followed in order for these four corner cells to contain the correct data. The z-direction (out of the page) follows the same logic, but for clarity is not shown.



CELL INDEXING

Figure 17. Cell indexing convention.

In order to enforce periodic boundary conditions on this cell, there are four update steps for this 2d example. These update steps are shown in Figure 18. Step one updates the ghost cells with values from the right side interiors ($NX, 1-NY$) as shown. Because interior values are used, these are correct. The second step takes the top interior values ($1-NX, NY$) along with ghost cell values at $(0, NY)$ and $(NX+1, NY)$. When transferred to the bottom ghost cell plane, cell 2 now has incorrect information as shown. To correct this cell, the third step is performed. This step takes the left interior values ($1, 1-NY$) along with ghost cell values at $(1, 0)$ and $(1, NY+1)$, and transfers these to the ghost cells on the right side. When step three has completed, the incorrect cell 2 value has been replaced with a correct value, but now cell 3 contains bad information as shown. Finally, the fourth step takes the bottom plane ($0-NX+1, 1$) and transfers these corrected values to the top ghost cells at $NY+1$. Now cell 3 has been replaced with correct data and all ghost cell information is correct. This exact procedure must be followed in this precise order to ensure that all ghost cells (especially the corners) are indeed updated with the correct information. For three dimensions (z-direction) would also have two additional update steps (after x and y), following the same logic.

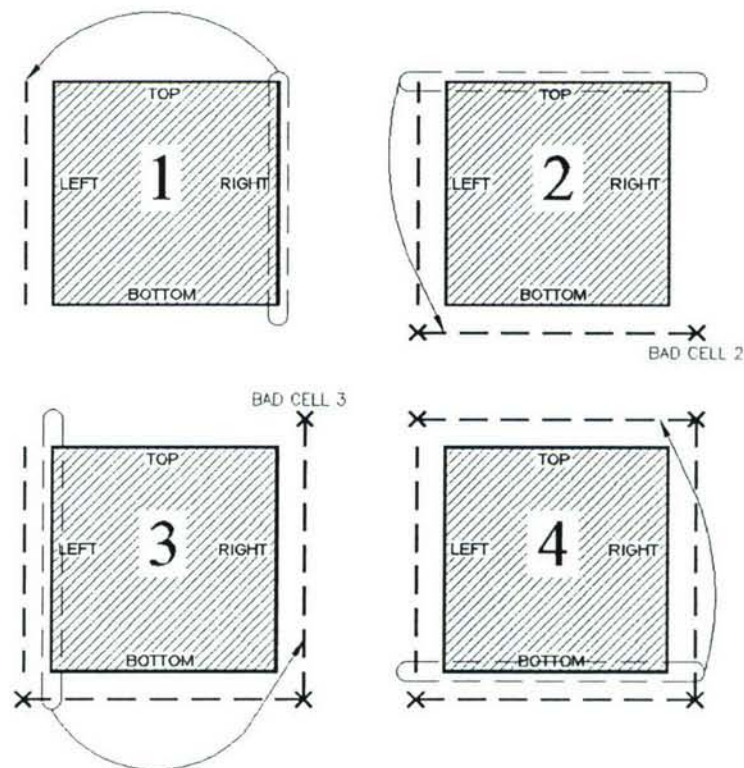


Figure 18. Ghost cell update procedure.

The Fortran code used for the the procedure in Figure 18 is given in Appendix C. For brevity only the x-direction will be explained in great detail, it really is trivial to understand and apply the same information for the y and z directions. The values of NX, NY, NZ are given by the number of interior points in the domain for each direction. The variable (xx) is a 3D array of data with correct interior domain information, but with incorrect boundary information stored at the ghost cells. The routines Send_BCX and Recv_BCX, are given in Appendix C. These are the procedures that update the ghost cell information.

```
Subroutine pPeriodic_BC(xx)

  Call Send_BCX(NX,NY,NZ,xx) !Send B/C info for x dir
  Call Recv_BCX(NX,NY,NZ,xx) !Recv B/C info for x dir
  Call Send_BCY(NX,NY,NZ,xx) !Send B/C info for y dir
  Call Recv_BCY(NX,NY,NZ,xx) !Recv B/C info for y dir
  Call Send_B CZ(NX,NY,NZ,xx) !Send B/C info for z dir
  Call Recv_B CZ(NX,NY,NZ,xx) !Recv B/C info for z dir

End Subroutine pPeriodic_BC
```

The routine above performs ghost cell updates 1 and 2 with Send/Recv_BCX. Ghost cell updates 3 and 4 are performed with Send/Recv_BCY. The send operation in Send_BCX uses the MPI function call MPI_ISEND and MPI_IRECV (commonly used MPI functions are shown in Appendix D). There are two basic ways to transfer information with MPI, these are blocking and non-blocking operations. A blocking operation (such as MPI_SEND) will perform the function call requested and also halts all processors from continuing any further until the requested process is completed. A non-blocking operation (such as a MPI_ISEND) will send the required data to all processors, but will not halt any processors from doing useful work while the send operation is in progress. This (will be shown shortly) can affect the overall performance of the code and care should be taken to decide which operation to perform. It was decided that the boundary conditions will use non-blocking send/receives and use MPI_WAITALL to guarantee this data is received in the appropriate order of Figure 18. The MPI_WAITALL command (Appendix D) allows the programmer to decide when a manual block should be placed in the code. The structure of the pPeriodic_BC routine uses MPI_ISEND and MPI_IRECV to send and receive the correct ghost cell boundary information. Because these are non-blocking, all of this data is sent at once to all processors. The routine Recv_BCX is just a MPI_WAITALL that will not let any processor continue until all of the send/receives for the x-direction are finished. Because non-blocking operations are performed, if the WAITALL incurs any performance penalty at all, it will be much less than performing a blocking send and a blocking receive. This same methodology is applied to the y and z directions. This strictly guarantees that the x-direction information is sent and received first, y-direction follows second, and the z direction is last.

The reader is now referred to the Send_BCX subroutine in Appendix C. The variables sent into this routine are the NX,NY,NZ, and the 3D array that is to be updated with periodic boundary conditions. A send buffer is created, (sbufx_r) that contains the values of the y-z plane at fixed interior x-location (beginning (1) or ending (NX)). To take out any ambiguity, the send buffer (x-direction) left plane and send buffer (x-direction) right plane are clearly shown in Figure 19, along with orientation axis.

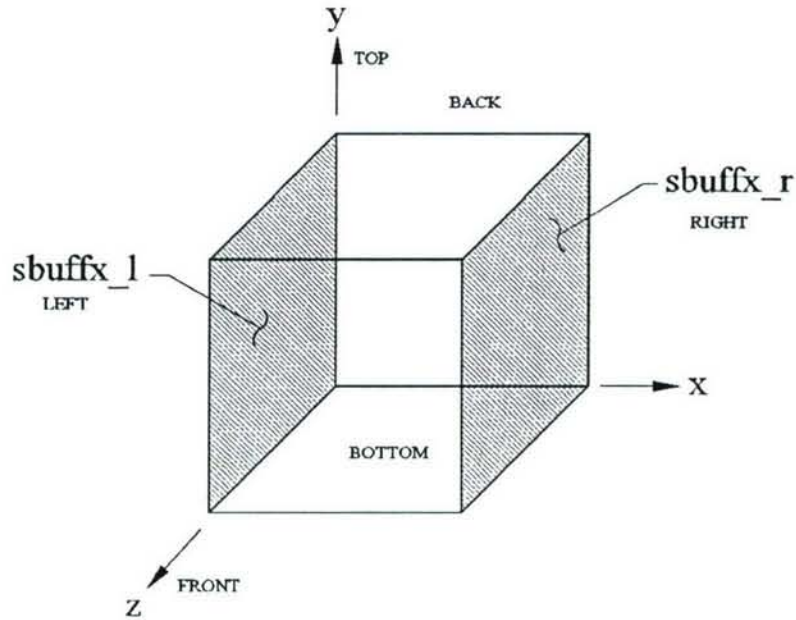


Figure 19. Orientation and planar data values used by periodic boundary conditions.

As is shown in Recv_BCX routine these x-direction planar values are now (once the operation is complete) used to update the ghost cell information for the input array that is desired to be periodic. For clarity these values are given by the receive buffer (x-direction) right side (rbuffx_r) and receive buffer (x-direction) left side (rbuffx_l). With the process explained in detail, it should be apparent that the y and z directions follow the exact same procedure outlined above, with only slight changes in orientation and by the naming convention of the send and receive buffers. For example the send buffer (y-direction) top x-z planar data is stored as sbuffy_t, and the sent bottom planar data is stored as sbuffy_b, etc.

5.4 MPI I/O

The CPU order shown in Figure 16 is determined by the input routine shown below.

```

      NXP = DNX/NX ! # processors in x-dir
      NYP = DNY/NY ! # processors in y-dir
      NZP = DNZ/NZ ! # processors in z-dir

! CPU location indices
      kl = myid/(NXP*NYP) + 1
      kr = mod(myid,NYP*NXP)
      jl = kr/NXP + 1
      il = mod(kr,NXP) + 1

! read in the total field and store in a temporary array
! called uJ.
      open (unit=503,file=trim(str4),form='formatted')
      read(503,*) ((uJ(i,j,k),i=0,DNX+1),j=0,DNY+1),k=0,DNZ+1)
!! now, let every processor get it's own "chunk" of data.
      tke(0:NX+1,0:NY+1,0:NZ+1) =
      uJ(NX*(il-1):(il*NX)+1,NY*(jl-1):(jl*NY)+1,NZ*(kl-1):(kl*NZ)+1
      )

      close(unit=503) !close the large field uJ.
```

The input routine is shown for the kinetic energy (tke). All other data fields (u, v, w, eps) are read in exactly the same way. In Fortran the expression $\text{mod}(n,m)$ gives the remainder when n is divided by m ; it is applied to integers. Examples are $\text{mod}(8,3) = 2$, $\text{mod}(27,4) = 3$, $\text{mod}(11,2) = 1$, $\text{mod}(20,5) = 0$. The number of processors in each direction are determined by dividing the total domain size (DNX, DNY, DNZ) into smaller problem sizes (NX, NY, NZ). Then integer division and the $\text{mod}(n,m)$ expression are used to compute three integer variables (il, jl, kl). The problem size plus boundaries is then defined for every processor and the code uses (il, jl, kl) to correctly grab a 'chunk' of data from the total domain. Finally the large (total) array uJ is closed.

Refer to the domain decomposition show in Figure 20, where a domain is split in half (x-direction). Assume (for this example) that we wish to run a 64x64x128 simulation on two processors as shown. The total domain is given as (DNX=64, DNY=64, DNZ=128), the problem size for each processor would be (NX=32, NY=64, NZ=128).

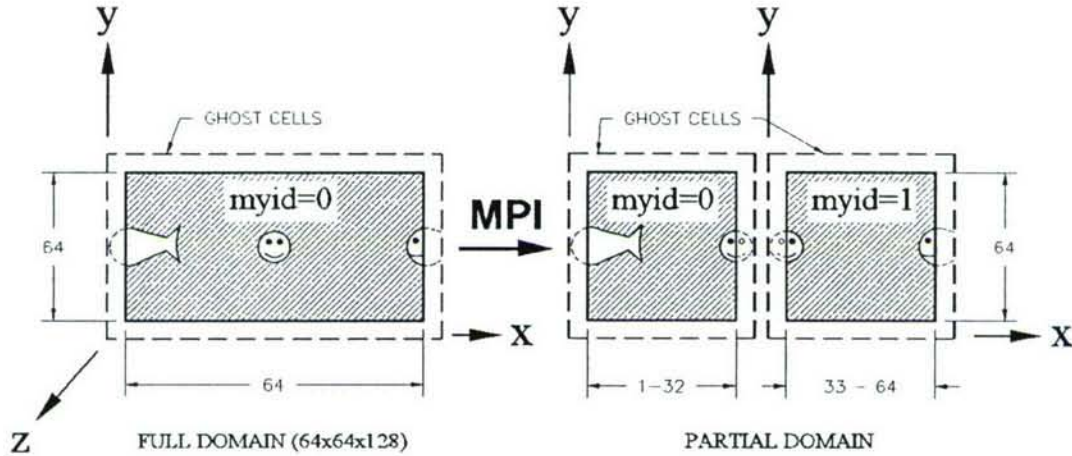


Figure 20. Single processor (left side) and MPI (right side) partitioning of the 64x64x128 example problem.

MPI will randomly assign the variable 'myid' to each processor it uses. In this case, two processors would be available with 'myid=0' and 'myid=1'. To ensure that the correct processor is located exactly as shown in Figure 20, the input routine above is used. This routine, reads in the total energy array (with boundaries) and stores this in a temporary array ($uJ(0:65,0:65,0:129)$) on each CPU. The processor with 'myid = 0', computes ($il=1, jl=1, kl=1$). This stores its 'chunk' of kinetic energy (tke) with information from the total domain $uJ(0:33, 0:65, 0:129)$, exactly as shown in Figure 20. Likewise, the processor 'myid = 1', computes ($il=2, jl=1, kl=1$), which stores its (tke) from the values $uJ(32:65, 0:65, 0:129)$. Finally, now that the large temporary array (uJ) has been partitioned correctly, it is closed. It is clear from this example, that these are indeed the correct indices for the input arrays. This guarantees that the input data will indeed follow the numbering sequence shown in Figure 18, with the appropriate interior domain and boundary (ghost cell) information.

5.5 Data Output

The output routine (Appendix F) follows exactly the same logic as the input routine, except in reverse order. Although this routine is shown for kinetic energy (tke), the other fields (u, v, w, eps) follow the same procedure. Typically a root processor is chosen, usually 'myid=0', is selected. This root processor will be the processor that collects all data from other CPUs using a (MPI_RECV) and pieces together the total domain by using the values of il, jl, kl . A simple 'If' statement allows all other processors (not root) to send (MPI_Send) their data along with the corresponding values (myid, il, jl, kl). Once the root processor has received all of the data from each processor and pieced together the total domain, it then writes out data for post-processing

use. The output routine shown does just this, and is clearly explained in Appendix F by the programmer comments.

5.6 Global Calculations

This last section deals with global variables used in the calculations of the code. Namely, these are the SUM, MINVAL, and MAXVAL Fortran calls. With the total domain now decomposed and sent to 'N' number of processors there must be a way to figure out the total SUM of data values globally (not just on each processor). Similarly the MINVAL/MAXVAL functions must give values for the total domain and not for each processor. In order to accomplish this, three extra routines were used in conjunction with the MPI_ALLREDUCE function (Appendix D). These three routines are shown in Appendix E. It should be straightforward to see that the ALLREDUCE function simply combines the values from all processors to calculate a desired global value. This global value is then distributed to all processors. For example, if an array is given by (1.0 2.0 3.0 4.0), and each value is sent to a parallel job with four processors, each processor stores the entries 1.0 - 4.0. If a Fortran SUM is performed, processor 1 calculates the total sum of the given array as 1.0, processor 2 calculates the total sum to be 2.0, etc. However with MPI_ALLREDUCE, the actual global sum of $(1.0 + 2.0 + 3.0 + 4.0 = 10.0)$ is calculated and this global value is then sent to every processor.

5.7 Parallel Optimization

With all the pieces mentioned above in place, the code was fully parallelized. Extensive testing and debugging proved that the parallel version of the code gives (almost to machine precision) the same solution as the serial version. The next and last step was to optimize the code in order to fully take advantage of using multiple processors. Optimization in parallel consists of efficiently sending, receiving, and computing data.

```

Start_MPI
!Non-Optimized
  Compute Sqrt(Data B)      !computes square root of Data
  MPI_SEND(Data A)          ! (blocking) send to all processors
                             (takes a while)
  !processors sit idle at this point until the send is
  !completed.
  Answer = (Data A and Sqrt(Data B))

!Optimized
  MPI_ISEND(Data A)          !non-blocking send to all
  processors (takes a while)
  Compute Sqrt(Data B)       !with send in progress,
  computer continues to work
  !once the Sqrt is calculated, the send has completed
  MPI_WAITALL                !manual stopping point to make
  sure Data A was sent
  Answer = (Data A and Sqrt(Data B))

End_MPI

```

A simple example is shown above, where a non-optimized and an optimized solution are computed with two arrays (Data A, Data B). The non-optimized code computes the square root of

(Data B) and then sends (Data A) to all processors. At this point, the answer cannot be computed until this sending operation has been completed. Recall that `MPI_SEND` is a blocking function call, and will halt all processors until the sending operation is completed. If (Data A) is very large, this send operation can take a long time, and at this point all processors sit idle and wait. When the send is completed the solution (Answer) is calculated. Obviously this will yield a correct answer, but it is not the best use of available resources because now an additional penalty is incurred by the communication time between processors.

The optimized code however, will yield the correct answer, and will do so faster! This is because the `MPI_ISEND` is performed (as shown). Recall that `MPI_ISEND` is a non-blocking operation. While this information is being sent, there is no reason why the processors cannot continue to do some useful work. As (Data A) is sent (optimized code), all processors begin computing the square root of (Data B). When the square root has been calculated, most (or ideally all) of (Data A) has been sent. If all of (Data A) was sent during the computation on (Data B), the `WAITALL` function incurs no communication penalty. Even if all of (Data A) was not sent, the `WAITALL` still takes less time than the blocking send performed by the non-optimized code. The final answer will be computed much faster. This hides the MPI communication times behind actual computational work in order to develop an ideally optimized code. By letting each processor do as much work as possible during these MPI communication function calls, better parallel performance is obviously obtained. This idea was mentioned above and shown to be effective for the periodic boundary condition routine shown in Section 4.1.2.

Figure 21 shows the total simulation time for a $128 \times 128 \times 256$ and a $64 \times 64 \times 128$ simulation before and after MPI optimization. This figure shows that once parallelized, the total simulation time for a $128 \times 128 \times 256$ simulation was 5.2 hours and 0.71 hours for the $64 \times 64 \times 128$ case, using eight processors. After the code was optimized by hiding as much of the communication time as possible, the total simulation time for both cases (again with 8 processors) dropped by roughly 20 percent. This reduction with the optimized code saves valuable supercomputing time.

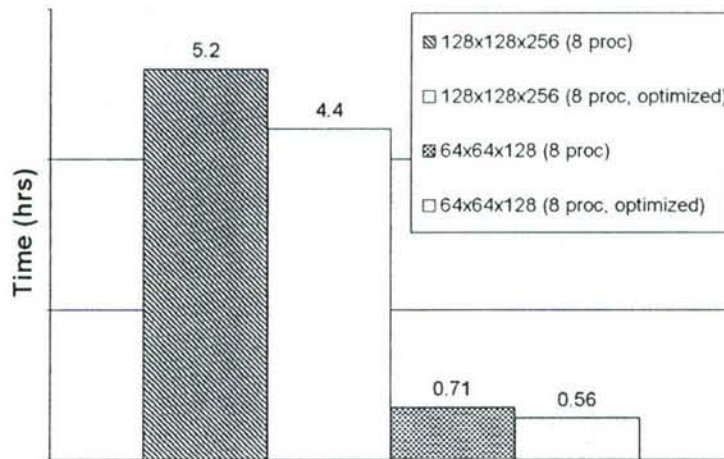


Figure 21. Total simulation time (hrs) for $64 \times 64 \times 128$ and $128 \times 128 \times 256$ before and after optimization on the UMass cluster.

5.8 MPI Performance

Figure 22 shows the parallel performance (log scale) as recorded on the UMass cluster named 'von Karman'. This figure shows the total simulation time (min) versus the number of CPUs used.

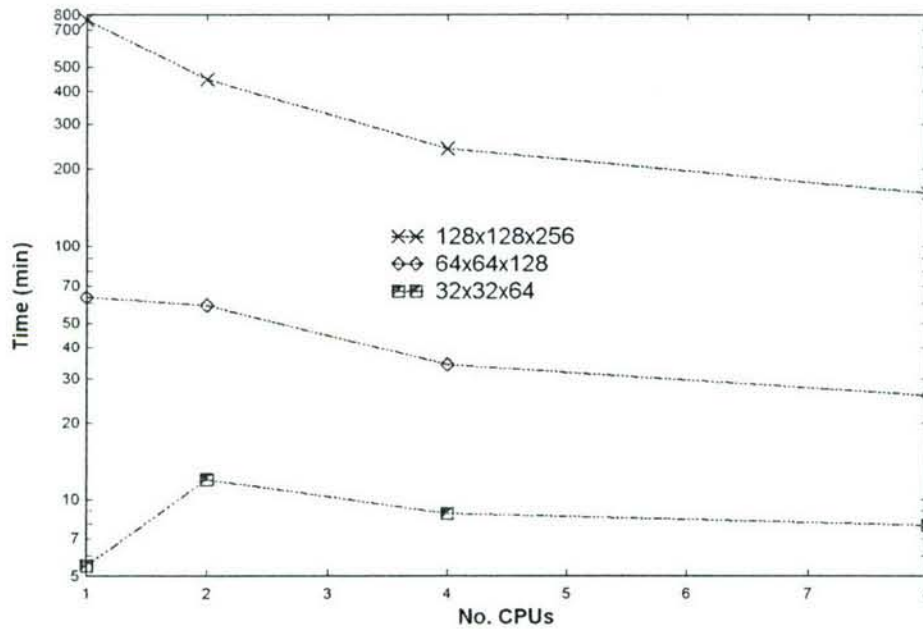


Figure 22. Total simulation time (min) on the UMass cluster vs. number of CPUs.

It is apparent that for the 32x32x64 simulation (squares), that using additional processors with von Karman increases the time to run a simulation. This is because the communication time between processors is greater than actual processor work. Luckily, the 64x64x128 (diamonds) and 128x128x256 (x's) have enough work on each processor so that the communication time is better hidden and is much lower. Here we see that using additional processors immediately decreases the simulation time. Because the UMass cluster was used extensively for testing and debugging, a plot of MPI performance such as Figure 22 was needed. It should be noted that von Karman is an old machine with a slow interconnect between processors. This is especially noticeable at the 32x32x64 mesh. A more accurate comparison of parallel performance is shown in Figure 23.

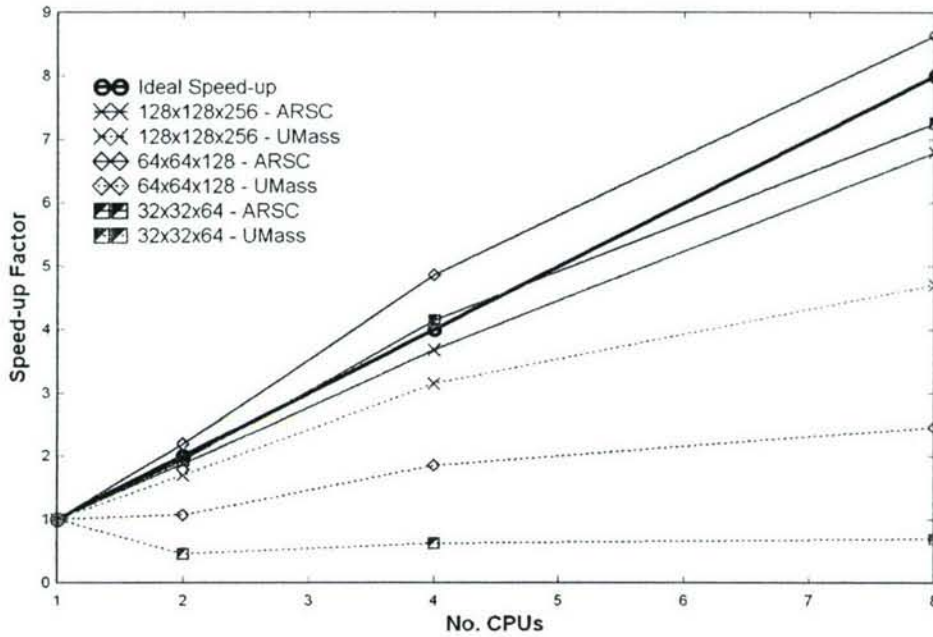


Figure 23. Speed-up factor vs. number of processors (UMass and ARSC).

This figure shows the parallel performance of the developed code as recorded on the Arctic Region Supercomputing center IBM/P690 named 'Iceberg' (symbols with solid lines) compared against 'von Karman' (symbols with dashed lines). The speed-up factor is given by the total simulation time on a single processor divided by the simulation time for multiple processors. It is not necessarily a measure of how fast a given CPU is, but rather, a measure of the communication time between processors for any given architecture. The ideal performance is also shown (solid line - circles) and shows a linear relationship. For example, using twice as many processors, should give (ideally) a speed-up factor of two, etc. Because von Karman has a slow interconnect, it is not surprising to see that it gives the worst performance (bottom three curves). Von Karman does not even begin to follow the ideal speed-up curve until the 128x128x256 mesh size. Even at this large mesh, the slow communication time between processors only gives a speed-up factor of 4.7 for eight processors. More interesting however, is how well the DoD funded supercomputer performs with the developed code. It has much faster communication between processors.

It is clear from Figure 23 increasing the number of processors on Iceberg immediately results in a faster simulation time. This is shown by (solid lines - symbols) the close behavior of the simulations on Iceberg to the ideal speed-up. Even with the small 32x32x64 simulation, Iceberg's communication time is so fast, that using additional processors immediately decreases simulation time (unlike von Karman). At the largest resolution of 128x128x256 (for eight processors) the speed-up on Iceberg is 6.8 compared to von Karman with 4.7, and the ideal of 8.0. As was expected, Iceberg outperforms von Karman. It should be mentioned that the increased speed-up factor for the 64x64x128 simulation on Iceberg has nothing to do with MPI or optimization. This has to do with the particular architecture the code is running on. In this case, Iceberg appears to

handle the 64x64x128 extremely well and gives better than ideal speed-up factors. With the available information on parallel performance, all of the large simulations 32x32x64 - 256x256x512 were performed with multiple processors at ARSC.

6 Reynolds Stress Transport Models

We wish to demonstrate that the ability to model turbulence at all mesh scales (universal modeling) is not a particular property of the modified (backscattering) k/ϵ model presented in section 4. It is a general approach that can work for many different RANS models. In this section we will show the behavior of a Reynolds stress transport (RST) model when it is used on a fine mesh as an LES subgrid scale model.

RST are the next level in complexity for turbulence modeling beyond two-equation models. They are particularly interesting in the context of this work because they can backscatter energy without any explicit modification. In particular, a modification such as α in the k/ϵ model is not necessary.

RST models contain more physics than two-equation models so they can lead to better predictions in the coarse mesh limit, and might even lead to better predictions at the LES mesh level. The exact Reynolds stress transport equation is,

$$\frac{DR_{ij}}{Dt} = \underbrace{P_{ij}}_{\text{exact}} + \underbrace{D_{ij}^v + D_{ij}}_{\text{exact}} - \underbrace{\epsilon_{ij}}_{\text{modeled}} + \underbrace{\Pi_{ij}}_{\text{modeled}} + \underbrace{\Omega_{ij}}_{\text{exact}} \quad (15)$$

This equation states that: the material derivative of R_{ij} equals the rate of production (P_{ij}), plus transport by diffusion ($D_{ij}^v + D_{ij}$), minus the rate of dissipation (ϵ_{ij}), plus redistribution due to turbulent pressure-strain interaction (Π_{ij}), plus redistribution due to rotation Ω_{ij} . The known quantities are labeled 'exact' and the unknown quantities require models to solve this system.

Major advantages with this approach are that the production term (usually a dominant term) is now exact, along with any rotational effects. Because transport equations are solved for the evolution of R_{ij} , the eddy viscosity hypothesis is not needed. Because an eddy viscosity assumption is not used, backscatter is possible. Finally, since most of the computational expense for this finite volume code is spent on solving a Poisson equation for pressure, additional transport equations may improve the accuracy of the model significantly, with only a minimal increase in computational requirements.

6.1 Universal RST Model

The modeled transport equation for the Reynolds stress tensor is,

$$\begin{aligned} \frac{\partial \bar{R}_{mn}}{\partial t} + \frac{\partial(u_j \bar{R}_{mn})}{\partial x_j} = \frac{\partial}{\partial x_j} \left[(\nu + \nu_\tau) \frac{\partial \bar{R}_{mn}}{\partial x_j} \right] + \bar{P}_{mn} - \bar{P} \bar{R}_{mn} + \\ Cp2^S (S_{mj} \bar{R}_{jn} + S_{nj} \bar{R}_{jm} + \bar{P} \bar{R}_{mn}) + Cp2^W (W_{mj} \bar{R}_{jn} + W_{nj} \bar{R}_{jm}) \\ + Cp1(\frac{\epsilon}{k})(\frac{2}{3} \delta_{mn} - \bar{R}_{mn}) + Cp2^* S_{mn} + 2\nu_\tau \left(\frac{1}{k} \frac{\partial k}{\partial x_k} \frac{\partial \bar{R}_{mn}}{\partial x_k} \right) \end{aligned} \quad (16)$$

This equation predicts the evolution of \bar{R}_{ij} where $\bar{R}_{ij} = \frac{R_{ij}}{k}$ is the dimensionless stress tensor. The model constants are given by, $Cp1 = 0.9$, $Cp2^S = 0.8$, $Cp2^W = 0.6$. The value of $Cp2^* = -0.2F^2$, where $F = \text{determinant}(\frac{3}{2} \bar{R}_{ij})$. The resolved strain rate tensor is defined as $S_{ij} = \frac{1}{2}(u_{i,j} + u_{j,i})$ and the resolved vorticity tensor is given by $W_{ij} = \frac{1}{2}(u_{i,j} - u_{j,i})$. The production term is given as $\bar{P}_{ij} = -(\bar{R}_{ik} u_{j,k} + \bar{R}_{jk} u_{i,k})$. The value of $\bar{P} = \frac{1}{2} \bar{P}_{ii}$, and the turbulent viscosity is defined as $\nu_\tau = \frac{C_\mu F k^2}{\epsilon}$. Note that the turbulent viscosity is only used in the diffusion terms. It is not used to determine the stresses.

Additionally, the transport equations for u_i , k and ϵ are again needed for closure and are shown below.

$$\frac{\partial u_i}{\partial t} + \frac{\partial}{\partial x_j} (u_i u_j) = -\frac{\partial(p + \frac{2}{3}k)}{\partial x_i} + \frac{\partial}{\partial x_j} [\nu \frac{\partial u_i}{\partial x_j} - k \bar{R}_{ij}] \quad (17)$$

$$\frac{\partial k}{\partial t} + \frac{\partial}{\partial x_j} (k u_j) = \frac{\partial}{\partial x_j} \left[\left(\frac{\nu + \nu_\tau}{\sigma_k} \right) \frac{\partial k}{\partial x_j} \right] + \bar{P} - \epsilon \quad (18)$$

$$\frac{\partial \epsilon}{\partial t} + \frac{\partial}{\partial x_j} (\epsilon u_j) = \frac{\partial}{\partial x_j} \left[\left(\frac{\nu + \nu_\tau}{\sigma_\epsilon} \right) \frac{\partial \epsilon}{\partial x_j} \right] + \frac{\epsilon}{k} [C_{\epsilon 1} \bar{P} - C_{\epsilon 2} \epsilon] \quad (19)$$

Once again, overbars on the velocity and pressure have been dropped for convenience. The production term (\bar{P}) is defined above. The constants are fairly standard k/ϵ constants, $C_{\epsilon 1} = 1.55$, $\sigma_\epsilon = 1.2$, $\sigma_k = 1.0$, $C_\mu = 0.15$. The parameter $C_{\epsilon 2}$ is determined from the analysis of Perot and de Bruyn Kops¹⁴.

6.2 Results

A number of simulations were performed with the RST model described above using the same isotropic decaying turbulence ($Re=640$) initial conditions used for the two-equation universal k/ϵ model. Simulations were performed for $32 \times 32 \times 64$ - $128 \times 128 \times 256$ mesh sizes and total kinetic energy predictions are shown in Figure 24. Once again the DNS data is shown by the large

circles. These VLES and LES simulations match the DNS data well. Small improvements might be possible by tuning some of the RST model constants.

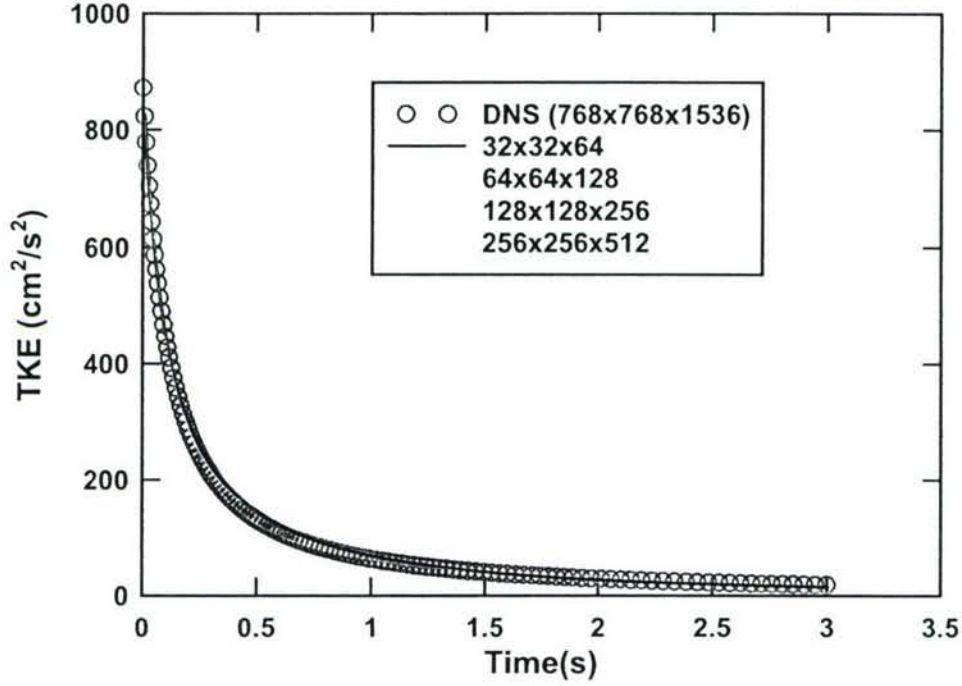


Figure 24. RST model total kinetic energy predictions for (initial $Re=640$) isotropic decaying turbulence.

Two of the kinetic energy ratios are shown in figure 25. These two mesh resolutions are the most difficult to capture since they are where the VLES behavior transitions to URANS. As with the previous k/ϵ model, the RST model maintains a relatively constant kinetic energy ratio, that tends to drift downwards after long times. This means that while the different meshes give the same total behavior (figure 24), they have different underlying solutions. The finer mesh cases automatically resolve more of the turbulence using first principals and model less of the turbulence. Unlike the k/ϵ version of this approach (that required a modification to enable energy backscatter) this RST model is identical to its RANS counterpart. It is simply being applied to LES meshes, and is shown to automatically perform as an LES subgrid scale model.

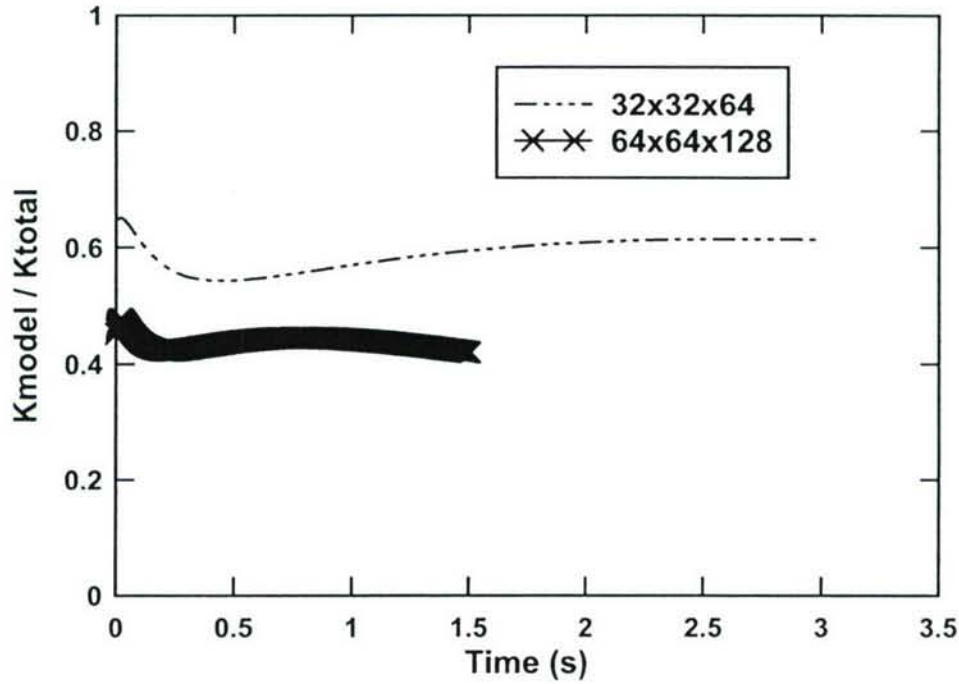


Figure 25. RST model ratio of modeled kinetic energy to total kinetic energy, Re=640.

7 Conclusions

The primary goal of this project was to demonstrate that RANS models can be used to model turbulence at any mesh resolution. This demonstration has been successfully performed using two very different types of RANS models. The key requirement of these models is that they must be able to backscatter energy. So some classical models which enforce a unidirectional energy transfer (such as simple eddy viscosity models) cannot automatically adapt to the available mesh resolution. The effectiveness of the RST model suggests that many algebraic Reynolds stress models and nonlinear k/ϵ models (that theoretically have the ability to backscatter) probably would perform like RST model and not require any backscatter modification.

We have demonstrated that this modeling approach automatically computes as much of the turbulence as the mesh allows, and models the rest. Different mesh resolutions produce the same total behavior, but produce very different underlying solutions. The finer the mesh, the more three-dimensional and unsteady the solution stays. Despite the fact that the equations are identical, the nonlinearity of the equation system means that solutions are sensitive to the mesh resolution. The method is robust. Starting with very different initial conditions, results in essentially the same solution behavior after long times. The length of time to adjust from arbitrary initial conditions appears to be proportional to the modeled turbulence eddy turn over time.

It was also demonstrated that this approach reproduces the classical LES length scale assumption (eddy length proportional to the mesh size) in the limit that that assumption is valid (in the inertial range). The advantage of the demonstrated approach is that it is valid much more broadly – both in the RANS and DNS limits, and in complex flows where an inertial range may well not exist.

The practical utility of a turbulence modeling approach that can be applied to any mesh cannot be overstated. This model is not dictated by the proximity of walls (like DES). It is not controlled by the user (like PANS), since it is rarely known to the user beforehand what kinetic energy ratio would be correct for a particular mesh. It does not blend an LES and RANS model together in an *ad hoc* way, like the other hybrid models. This approach is, in many ways, much simpler. We have demonstrated that appropriately chosen RANS models can function well at any mesh resolution. These models lead to a universal modeling approach that can be applied at any mesh resolution. The user specifies what they can afford, and the model provides the best solution it can with what the user can afford.

This approach has the potential to make LES far more accessible. Existing commercial and industrial CFD codes with RANS models can very easily do LES now. A new code and model is not necessary. New expertise is not necessary.

Finally, this project has broken the artificial wall that had been built between the LES and RANS modeling approaches. Despite the differences in terminology, and the insistence of various model developers to the contrary, this work unambiguously shows there is no fundamental distinction between the RANS and LES modeling approaches.

8 Bibliography

- [1] Carati, D., Ghosal, S., and Moin, P. On the representation of backscatter in dynamic localization models. *Physics of Fluids* 07 (1995), 606–616.
- [2] Chang, W., Giraldo, F., and Perot, J.B. Analysis of an exact fractional step method. *Journal of Computational Physics* 180 (2002), 183–199.
- [3] Chasnov, J. Simulation of the kolmogorov inertial subrange using an improved subgrid model. *Physics of Fluids A* 3 (1991), 188–200.
- [4] Comte-Bellot, G., and Corrsin, S. Simple eulerian time correlation of full and narrow-band velocity signals in grid-generated isotropic turbulence. *Journal of Fluid Mechanics* 48 (1971), 273–337.
- [5] de Bruyn Kops, S.M., and Riley, J.J. Direct numerical simulation of laboratory experiments in isotropic turbulence. *Physics of Fluids* 10 (1998), 2125–2127.
- [6] de Bruyn Kops, S.M., Riley, J.J., and Kos'aly, G. Direct numerical simulation of reacting scalar mixing layers. *Physics of Fluids* 13 (2001), 1450–1465.
- [7] Deardorff, J.W. The use of subgrid transport equations in a three-dimensional model of atmospheric turbulence. *ASME J. Fluids Engineering* 95 (1973), 429.
- [8] Germano, M. Turbulence: the filtering approach. *Journal of Fluid Mechanics*, **238** (1992), 325–336.

- [9] Ghosal, S., Lund, T.S., Moin, P., and Akselvoll, K. A dynamic localization model for large-eddy simulation of turbulent flows. *Journal of Fluid Mechanics*, 286 (1995), 229–255.
- [10] Girimaji, S., Sreenivasan, R., and Jeong, R. Pans turbulence model for seamless transition between rans,les: fixed-point analysis and preliminary results. In *Proceedings of ASME-JSME Joint Fluids Engineering Conferences* (2003).
- [11] Harlow, F.H., and Welch, J.E. Numerical calculation of time dependent viscous incompressible flow of fluid with free surface. *Physics of fluids* 8 (1965), 2182.
- [12] Jones, W.P., and Launder, B.E. The prediction of laminarization with a two-equation model of turbulence. *International Journal of Heat and Mass transfer* 15 (1972), 301–314. 62
- [13] Nikitin, N.V., Nicoud, F., Wasistho, B., Squires, K.D., and Spalart, P.R. An approach to wall modeling in large-eddy simulations. *Physics of Fluids* 12 (2000), 1629–1632.
- [14] Perot, J.B., and de Bruyn Kops, S. Modeling turbulent dissipation at low and moderate Reynolds numbers. *Journal of Turbulence* 07 (2006).
- [15] Perot, J. B. & Chartrand, C., *Modeling Return to Isotropy Using Kinetic Equations*, Physics of Fluids. 17 (3), 2005.
- [16] Piomelli, U., Balaras, E., Pasinato, H., Squires, K., and Spallart, P. The innerouter layer interface in large-eddy simulations with wall-layer models. *International Journal of Heat and Fluid Flow* 24 (2003), 538–550.
- [17] Schumann, U. Subgrid scale model for finite difference simulations of turbulent flows in plane channels and annuli. *Journal of Computational Physics* 18 (1975), 376–401.
- [18] Smith, B.R. A near wall model for the k-l two equation turbulence model. *AIAA* 94-2386 (1994).
- [19] Spalart, P., and Allmaras, S. A one-equation turbulence model for aerodynamic flows. *Recherche Aerospaciale* 1 (1994), 5–21.
- [20] Spalart, P., Jou, W., Streetlets, M., and Allmaras, S. Comments on the feasibility of les for wings, and on a hybrid rans/les approach. *First AFOSR International Conference on DNS/LES, Ruston, Louisiana, USA* (2001).
- [21] Speziale, C.G. Turbulence modeling for time-dependant rans and vles: A review. *AIAA Journal* 36 (1998), 173–184.
- [22] Wilcox, D.C. *Turbulence modeling for CFD*. La Canada, CA: DCW Industries, 1993.
- [23] Perot, J. B. & Gadebusch, J., A Self-adapting Turbulence Model for Flow Simulation at any Mesh Resolution. Submitted to Phys. Fluids, Jan 2007.
- [24] Gadebusch, J. , On the development of self-adapting (RANS/LES) turbulence models for fluid simulation at any mesh resolution, Master Thesis, University of Massachusetts, Amherst, 2007.

9 Appendices

A. Derivation of the Evolution Equations

The incompressible Navier-Stokes equations are:

$$u_{k,k} = 0 \quad (7)$$

$$u_{i,i} + (u_i u_k)_{,k} = -p_{,i} + \sigma_{ik,k} \quad (8)$$

Following the ideas of Germano, the moment of (8) is taken with u_j , another moment of this same equation with indices interchanged yields (9) and (10).

$$u_j u_{i,i} + u_j (u_i u_k)_{,k} = -p_{,i} u_j + u_j \sigma_{ik,k} \quad (9)$$

$$u_i u_{j,i} + u_i (u_j u_k)_{,k} = -p_{,j} u_i + u_i \sigma_{jk,k} \quad (10)$$

Adding together (9) and (10) and using a little calculus, it can be shown that the resulting equation can be written as:

$$(u_i u_j)_{,i} + (u_i u_j u_k)_{,k} = - \left[p u_i \delta_{jk} + p u_j \delta_{ik} - \nu (u_i u_j)_{,k} \right]_{,k} + 2 p s_{ij} - 2 \nu u_{i,k} u_{j,k} \quad (11)$$

if a Newtonian fluid $\sigma_{ij} = \nu u_{i,j}$ is assumed. Recall that the turbulent stress tensor was previously defined as $R_{ij} = \overline{u_i u_j} - \overline{u_i} \overline{u_j}$, the double bracket was $\langle a_i, b_j \rangle \equiv \overline{a_i b_j} - \overline{a_i} \overline{b_j}$, and the turbulent transport term was defined by $T_{ijk} \equiv \overline{u_i u_j u_k} - \overline{u_i} \overline{u_j} \overline{u_k} - \overline{u_i} R_{jk} - \overline{u_j} R_{ik} - \overline{u_k} R_{ij} - \overline{u_i} \overline{u_j} \overline{u_k}$. Simply averaging equations (7), (8), and (11) and substituting for the stress tensor, double bracket, and turbulent transport will recover the equations below:

$$\overline{u_{k,k}} = 0 \quad (12)$$

$$\overline{u_{i,i}} + \overline{(u_i u_j)_{,j}} = -\overline{p_{,i}} + \nu \overline{u_{ij,j}} - R_{ij,j} \quad (13)$$

$$\begin{aligned} R_{ij,i} + \overline{u_k} R_{ij,k} &= \nu R_{ij,kk} - (R_{jk} \overline{u_{i,k}} + R_{ik} \overline{u_{j,k}}) \\ &\quad - \langle T_{ijk} \rangle_{,k} - (\langle p_{,i}, u_j \rangle + \langle p_{,j}, u_i \rangle) - 2\nu \langle u_{i,k}, u_{j,k} \rangle \end{aligned} \quad (14)$$

Hence it is shown that the averaging invariance procedure of Germano does indeed provide evolution equations that are exactly the same as the Reynolds stress transport (RST) equations.

B. CPU Neighbors

```
Subroutine CPU_neigh()
Implicit None
Integer :: iz,iy,ix, tmp
```

```

!!location in CPU grid
iz = myid/(NXP*NYP) + 1
tmp = myid-(iz-1)*NXP*NYP
iy = tmp/NXP + 1
ix = tmp -(iy-1)*NXP + 1

!Compute neighboring CPU locations below

CPU_info(1) = myid+1 !x-dir right
if ( ix == NXP) CPU_info(1) = CPU_info(1) - NXP

CPU_info(2) = myid-1 !x-dir left
if ( ix == 1 ) CPU_info(2) = CPU_info(2) + NXP

CPU_info(3) = myid+NXP !y-dir top
if ( iy == NYP) CPU_info(3) = CPU_info(3) - NXP*NYP

CPU_info(4) = myid-NXP !y-dir bottom
if ( iy == 1 ) CPU_info(4) = CPU_info(4) + NXP*NYP

CPU_info(5) = myid+NXP*NYP !z-dir back
if ( iz == NZP) CPU_info(5) = CPU_info(5) - NXP*NYP*NZP

CPU_info(6) = myid-NXP*NYP !z-dir front
if ( iz == 1 ) CPU_info(6) = CPU_info(6) + NXP*NYP*NZP

End Subroutine CPU_neigh

```

Integer 'myid', is a variable MPI assigns to each processor in use. If eight processors are used in parallel 'myid' would range from 0 - 7. The variables NXP, NYP, and NZP correspond to the number of processors in each direction (x,y,z). The calculated integer, CPUinfo1, is the CPU located to the right of the processor in question. CPUinfo2 computes the CPU to the left of the processor in question. CPUinfo3 and CPUinfo4 correspond to the top and bottom CPUs respectively. Lastly, CPUinfo5 and CPUinfo6 are the back and front CPUs located relative to the CPU in question. Further clarification by (top, bottom, etc.) was clearly shown in Figure 16.

C. Periodic Boundary Condition Data Transfer

```

! Nonblocking MPI send x
Subroutine Send_BCX(NX,NY,NZ,var)
Implicit None
Integer :: NX,NY,NZ
Real, Intent(IN) :: var(0:NX+1,0:NY+1,0:NZ+1)
Integer :: num, tag1,tag2, cpu1,cpu2
Real :: tmp1,tmp2

num = (NY+2)*(NZ+2)
cpu1 = CPU_info(1)
cpu2 = CPU_info(2)
tag1 = tag+1
tag2 = tag+2

!Sending right y-z plane
sbufx_r(0:NY+1,0:NZ+1) = var(NX,0:NY+1,0:NZ+1)

```

```

call mpi_isend (sbufx_r,num,datasize,cpul,tag1,MPI_COMM_WORLD,
               handle_sx(1),ierr)

!Sending left y-z plane
sbufx_l(0:NY+1,0:NZ+1) = var(1,0:NY+1,0:NZ+1)
call mpi_isend (sbufx_l,num,datasize,cpu2,tag2,MPI_COMM_WORLD,
               handle_sx(2),ierr)

!Receiving right y-z plane
call mpi_irecv (rbuffx_r,num,datasize,cpul,tag2,MPI_COMM_WORLD,
               handle_rx(1),ierr)

!Receiving left y-z plane
call mpi_irecv (rbuffx_l,num,datasize,cpu2,tag1,MPI_COMM_WORLD,
               handle_rx(2),ierr)

End Subroutine Send_BCX

*****
! Nonblocking MPI recieve - x
Subroutine Recv_BCX(NX,NY,NZ,var)
Implicit None
Integer :: NX,NY,NZ , i
Real :: var(0:NX+1,0:NY+1,0:NZ+1)

! wait for recv and send to complete
call mpi_waitall(2,handle_sx,status_s,ierr) !wait for sends
call mpi_waitall(2,handle_rx,status_r,ierr) !wait for recvs

!from the left (high #)
!move data values from buffers into ghost cells
var(0,0:NY+1,0:NZ+1) = rbuffx_l(0:NY+1,0:NZ+1)

!from the right CPU (low #)
!move data values from buffers into ghost cells
var(NX+1,0:NY+1,0:NZ+1) = rbuffx_r(0:NY+1,0:NZ+1)

End Subroutine Recv_BCX

```

D. Typical MPI Functions Used

Great detail explaining MPI function calls can be found at: <http://www-unix.mcs.anl.gov/>
Below are a selection of the main function calls used in this project.

```

*****
MPI_Isend( void *buf, int count, MPI_Datatype datatype, int dest, int tag,
           MPI_Comm comm, MPI_Request *request )

MPI_Irecv( void *buf, int count, MPI_Datatype datatype, int source,
           int tag, MPI_Comm comm, MPI_Request *request )

Input Parameters:
buf:          initial address of send/recieve buffer (choice).

```

count: number of elements in send/receive buffer (integer).
 datatype: datatype of each send/receive buffer element (handle).
 source: rank of source (integer).
 dest: rank of destination (integer).
 tag: message tag (integer).
 comm: communicator (handle).

Output Parameter

request: communication request (handle)

All MPI routines in Fortran have an additional argument ierr at the end of the argument list.

MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,
 int tag, MPI_Comm comm)

MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
 int tag, MPI_Comm comm, MPI_Status *status)

Input Parameters:

buf: initial address of send/receive buffer (choice)
 count: number of elements in send buffer (nonnegative integer)
 datatype: datatype of each send/receive buffer element (handle)
 dest: rank of destination (integer)
 tag: message tag (integer)
 comm: communicator (handle)

Output Parameters:

buf: initial address of receive buffer (choice)
 status: status object (Status)

MPI_Waitall(
 int count,
 MPI_Request array_of_requests[],
 MPI_Status array_of_statuses[])

Input Parameters:

count: lists length (integer).
 array_of_requests: array of requests (array of handles)

Output Parameter: array_of_statuses

array of status objects (array of Status). May be MPI_STATUSES_IGNORE

MPI_Allreduce (void *sendbuf, void *recvbuf, int count,
 MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

Input Parameters:

sendbuf: starting address of send buffer (choice)
 count: number of elements in send buffer (integer)
 datatype: data type of elements of send buffer (handle)
 op: operation (handle)
 comm: communicator (handle)

Output Parameter:

recvbuf:
 starting address of receive buffer (choice)

E. Global Values

```
Real Function pSUM(var)
Implicit None
Real :: num, var(:, :, :)
Real :: data_G

num = SUM(var)

if (numprocs > 1) then
Call MPI_Allreduce(num, data_G, 1, datasize, MPI_SUM, MPI_COMM_WORLD,
                   ierr) !MPI_REAL
    pSUM = data_G
else
    pSUM = num
end if

End Function pSUM

*****
Real Function pMAXVAL(var)
Implicit None
Real :: num, var(:, :, :)
Real :: data_G

num = MAXVAL(var)

if (numprocs > 1) then
Call MPI_Allreduce(num, data_G, 1, datasize, MPI_MAX, MPI_COMM_WORLD,
                   ierr) !MPI_REAL
    pMAXVAL = data_G
else
    pMAXVAL = num
end if

End Function pMAXVAL

*****
Real Function pMINVAL(var)
Implicit None
Real :: num, var(:, :, :)
Real :: data_G

num = MINVAL(var)

if (numprocs > 1) then
Call MPI_Allreduce(num, data_G, 1, datasize, MPI_MIN, MPI_COMM_WORLD,
                   ierr) !MPI_REAL
    pMINVAL = data_G
else
    pMINVAL = num
end if

End Function pMINVAL
```

F. Data Output

```

Subroutine TKE_OUT(dir)
Implicit None
Real    :: uJ(0:DNX+1,0:DNY+1,0:DNZ+1)
Real,dimension(0:NX+1,0:NY+1,0:NZ+1) :: rbuff
Real    :: intsend(1:3),intrecv(1:3)
Integer :: sendcount,recvcount,root,tag,tag2,ierr,ss(MPI_STATUS_SIZE)

!(NXP, NYP, NZP, il, jl, kl) are computed exactly as in input
root = 0 !assign root processor to be myid = 0.
recvcount = ((NX+2)*(NY+2)*(NZ+2)) !size of data to be sent
sendcount = recvcount !size of data to be received.

If (myid == 0) Then

!Store root processor data section into larger array (uJ).
uJ( NX*(il-1):(il*NX)+1 , NY*(jl-1):(jl*NY)+1, NZ*(kl-1):(kl*NZ)+1 ) =
tke(0:NX+1, 0:NY+1, 0:NZ+1)

!After storing root information, receive the values (myid, il, jl, kl)
!from all other processors

do i = 1, numprocs-1 !loop over all processors (except root)

!Now receive data from processor "i"
Call MPI_RECV(rbuff,recvcount,datasize,i,tag,MPI_COMM_WORLD,ss,ierr)

!Now receive (il,jk,kl) from the processor that just sent data
Call MPI_RECV(intrecv,3,datasize,i,tag2,MPI_COMM_WORLD,ss,ierr)
!with myid known, along with (il,jl,kl) reconstruct the large array (uJ)
!from the receive buffer.
uJ( NX*(il-1):(il*NX)+1 , NY*(jl-1):(jl*NY)+1, NZ*(kl-1):(kl*NZ)+1 )
= rbuff(0:NX+1, 0:NY+1, 0:NZ+1)

end do !stops loop on root

!!!! now write out total domain to binary file.
open (unit=500,file=trim(str1),form='binary')
write(500) ((uJ(i,j,k),i=1,DNX),j=1,DNY),k=1,DNZ)
close(unit=500)

Else

!If processor is not root, then send your tke data to root.
Call MPI_SEND(tke,sendcount,datasize,root,tag,MPI_COMM_WORLD,ierr)
!store il,jl,kl in an array called (intsend)
intsend(1) = real(il)
intsend(2) = real(jl)
intsend(3) = real(kl)

!Now send values of (il,jl,kl) to root.
Call MPI_SEND(intsend,3,datasize,root,tag2,MPI_COMM_WORLD,ierr)

End If

Return
End Subroutine TKE_OUT

```